The magazine of the ACCU

www.accu.org

{cvu}

Volume 21 Issue 6 January 2010 £3

Features

{cvu}

Volume 21 Issue 6
January 2010
ISSN 1354-3164
www.accu.org

Features Editor
Steve Love
cvu@accu.org

Regulars Editor
Jez Higgins
jez@jezuk.co.uk

Contributors
Giovanni Asproni, Frances Buontempo, Pete Goodliffe, Paul Greyner, Richard Harris, Jon Jagger, Derek Jones, Alan Lenton, Alison Lloyd, Roger Orr, Joe Wood

ACCU Chair
Jez Higgins
chair@accu.org

ACCU Secretary
Alan Bellingham
secretary@accu.org

ACCU Membership
Mick Brooks
accumembership@accu.org

ACCU Treasurer
Stewart Brodie
treasurer@accu.org

Advertising
Seb Rose
ads@accu.org

Cover Art
Pete Goodliffe

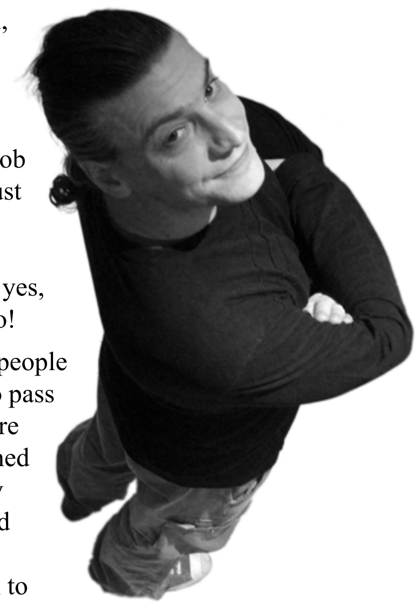
Repro/Print
Parchment (Oxford) Ltd

Distribution
Able Types (Oxford) Ltd

Design
Pete Goodliffe

accu

Tales From the Other Side



Good grief! 2010 and I'd barely got used to it being 2009! Anyway, a very Happy New Year to you all, I hope we've enough in this issue to stir the brain cells after any recent festivities that might have occurred.

At the time of writing this, I'm preparing to leave one job and start another almost immediately afterward, with just one as-ever-too-short weekend betwixt. All of which means my professional life is a muddle of rushed explanatory telephone conversations, meetings and, oh yes, writing documents that should've been written ages ago!

Handing over the work of months or years to different people can be a distressing and sobering activity. You'd like to pass information on in a way you'd like to receive it; I'm sure many of us have been on the receiving end of such rushed documentation and vague meetings, leaving us in every possible doubt about what – if anything – we've learned from the experience. And then, having to do all the groundwork again, only to ultimately have to pass it on to someone else...

Just deciding on which bits are important enough to document, and correspondingly, which are not, can make a great deal of difference between useful documentation and unintelligible nonsense.

There are striking parallels here to writing and reading code; you pick up someone else's code, which they presumably thought was brilliant, and can make neither head nor tail of it. Resorting at last to making the comments visible, you discover that they're not much help either. Finally, you go trawling through the archives of the version control system to find the culprit...eventually discovering the awful truth: not only does the person in question work nearby (either ***still*** or ***again***), you know them rather well...

Steve

STEVE LOVE
FEATURES EDITOR

The official magazine of ACCU

ACCU is an organisation of programmers who care about professionalism in programming. That is, we care about writing good code, and about writing it in a good way. We are dedicated to raising the standard of programming.

ACCU exists for programmers at all levels of experience, from students and trainees to experienced developers. As well as publishing magazines, we run a respected annual developers' conference, and provide targeted mentored developer projects.

The articles in this magazine have all been written by programmers, for programmers – and have been contributed free of charge.

To find out more about ACCU's activities, or to join the organisation and subscribe to this magazine, go to www.accu.org.

Membership costs are very low as this is a non-profit organisation.

DIALOGUE

REGULARS

FEATURES

SUBMISSION DATES

C Vu 22.1:	1st February 2010
C Vu 22.2:	1st April 2010
Overload 96:	1st March 2010
Overload 97:	1st May 2010

WRITE FOR C VU

Both C Vu and Overload rely on articles submitted by you, the readers. We need articles at all levels of software development experience. What are you working on right now? Let us know!

Send articles to cvu@accu.org. The friendly magazine production team is on hand if you need help or have any queries.

ADVERTISE WITH US

The ACCU magazines represent an effective, targeted advertising channel. 80% of our readers make purchasing decisions or recommend products for their organisations.

To advertise in the pages of C Vu or Overload, contact the advertising officer at ads@accu.org.

Our advertising rates are very reasonable, and we offer advertising discounts for corporate members.

COPYRIGHTS AND TRADE MARKS

Hunting the Snark (Part 5)

Alan Lenton investigates engineering in software.

Hmm – somehow I got overlooked last issue (such things are the story of my life...). However, those of you with long memories will recall that in the issue before last I started looking at the question of whether ‘software engineering’ is a true branch of engineering, and if not, why not. In that article we looked at contributions on this from the BCS (more of an assertion than any justification), Tom Demarco, and Chuck Connell. In this article I want to look at a piece from Koni Buhrer [0]

Buhrer argues that the difference between a craft and engineering is that craft is based on trial and error, while engineering is based on a set of first principles (usually the laws of physics) from which all knowledge in the discipline can be derived. Note that this does not involve any concept of ‘maturity’ (however you want to define that). I would suggest, contrary to the views of the BCS, that maturity is probably necessary for a move from a craft to engineer, but it is far from a sufficient requirement.

Buhrer takes a hard line on the issue of software as engineering. Not only does he believe that current software development is a craft and not engineering, he is of the view that software development will never be an engineering discipline! As he points out, although we have rules and methods for how we go about software development (think ‘best practices’) they are not rules derived from first principles, they are something that has come out of collective experimentation.

Let me give you an example. If you give a qualified civil engineer fresh out of college a building to design, the result may well look like something that was ripped out of the Maginot line, but it won’t fall down, and the engineer will be able to prove that it won’t fall down from first, physical, principles. A more experienced engineer would probably design something more elegant, and probably more economic to build, but they too would be able to prove that it wouldn’t fall down [1].

Contrast that with even an experienced programmer. How many people do you know who can prove, from some sort of first principles, not using tests, that a program of modest complexity will work as intended?

The interesting thing about Buhrer’s paper is that he has an explanation for why software development is so different and so much more difficult. If you look at engineering processes, there are basically two stages – design and manufacture. Buhrer’s thesis is that software development is design, and that the manufacture is the running of the compiler and linker. I suspect that this idea was developed in an enterprise development environment, and would need some re-working for shrink wrap.

> craft is based on trial and error, while engineering is based on a set of first principles

None-the-less, the idea gives some intriguing answers to a number of common problems in software development. Take for instance the difficulty in providing a detailed work breakdown and schedule for software development, in the way you can for, say building a skyscraper. To quote Buhrer, ‘Because software implementation is the equivalent of skyscraper design (i.e., the creation of the blueprints for a skyscraper) and not skyscraper construction. Design work is naturally less amenable to planning than construction work – which is true for skyscrapers as well as for software – because the scope and complexity of the end product are discovered only in the course of the design work’.

It also explains the lack of economies of scale in software programs. I’m sure my readers will be only too well aware that if you increase the required size of a program by (say) ten times, it costs way more than ten times to produce. That’s because, in engineering, the economies of scale come from the economies in the manufacturing cost, not the design stages, which will, probably, take proportionally longer!

Furthermore, the construction is where most of the cost is in engineering – building the skyscraper costs far more than designing the skyscraper, and so the cost benefits of scale are realised in the building (manufacturing), not the design.

> commercial programming may become tied up in regulation

It is interesting to speculate whether civil engineering would have emerged from craft if it had been possible to press a button on a blueprint and have the building appear within a few hours at no cost, and facilities to dispose of it at the press of a button! Without the need to have to prove the design before the bulk of the upfront costs were committed, would civil engineering have developed, or would the craft have just proceeded down the route of design, construct, test, demolish, tweak design, construct...

Buhrer’s paper goes further, and tries to provide, with much less success, the idea of some first principles. Unfortunately, the exposition is left to a part two of the article, which I have been unable to trace, so I guess it may never have been developed or published.

OK. So why is all this important?

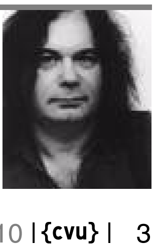
It’s important because the bulk of the scientific breakthroughs in the last 20 years have been enabled by the rise of cheap computing power. At the same time, software development appears to be little further advanced than that of a medieval craft guild state. That, rightly, concerns its practitioners [2].

At the same time as we are struggling to understand the basis of our discipline, powerful forces are moving to place limits on our activities. Governments are not happy with us. Spectacular failures of government contracts (mostly their own fault, but what government would admit that), combined with the fact that software development is a very portable skill, making it easy to pack and leave the country, is providing the impetus for registration and regulation of commercial programmers. It seems likely that my generation will be the last of the self-taught programmers.

On the bright side, while commercial programming may become tied up in regulation, it will be difficult to completely stop ‘amateur’ programming. After all as long as open source software exists, all you need is a general purpose computer, an editor and a compiler to write your own programs! And if you break the program, it’s not as though you have to buy ones and zeros from the shop to fix it...

I think we are on the cusp of something big which will determine the way in which our profession will develop for many years to come. Programming is now not something separate from society, if it ever was, it genuinely is becoming ubiquitous, and we can no longer shelter behind geekiness. We can either set our own course, which means we need

ALAN LENTON



Alan is a programmer, a sociologist, a games designer, a wargamer, writer of a weekly tech news and analysis column, and an ocassional writer of short stories (see http://www.ibgames.com/alan/crystalfalls/index.html if you like horror). None of these skills seem to be appreciated by putative employers...

A Brief Introduction to F#

Joe Wood shares his experiences with a new .Net functional language.

F# [1] is a new multi-paradigm programming language (with a strong functional programming leaning) developed at Microsoft Research in Cambridge under the leadership of Don Syme for the .NET environment. This is not intended to be a tutorial on F#, merely something to whet your interest and point you in the direction of more comprehensive material.

Before we begin, I should point out that I am a neither an F# aficionado nor a functional programming wizard, barely an apprentice. All of my professional programming has been in mainstream imperative languages of the ALGOL tradition. My doctoral project used pop-11 [2] which is basically Lisp and Forth meet the ALGOL style languages, so this is a learning experience for us all.

Why would you be interested in F#?

It is a good question. It takes time and effort to study a new language and the pay back may not be quick. There are two separate answers to this question.

The basic answer is that it expands our knowledge and gives us new ways of looking at things and therefore provides new insights to our everyday problems.

More specifically, F# is a multi-paradigm language derived from Ocaml, which supports functional programming, object-oriented programming and plain old imperative programming. It is strongly typed, but the compiler can and does deduce most of the types. Strict functional programming means that there are no side effects and hence it is ‘easier’ to partition into independent pieces to run on separate cores in multi-core chips, [1 (chap. 13)]. F# will be supported by Visual Studio 2010 as a first-class language, but its status in Visual Studio Express is less certain.

F# is primarily a compiled language, but also provides an interactive interpreter which is useful for rapid proto-typing. In Visual Studio you can invoke the interpreter on highlighted code fragments using **`alt-enter`**.

Functional programming?

If you have never come across functional programming languages it is reasonable to ask: what is it all about? In imperative languages the focus is on data and how to manipulate it, in a blow by blow manner. For example, suppose you want to find the sum of some data, a C style solution is:

```
int sum ( const int * data, const size_t len ) {
  int total = 0;
  for (size_t i=0; i<len; i++) {
      total += *(data+i) ;
  }
  return total ;
}
```

The important points are that we have to keep track of our position in **`data`** and that **`total`** keeps changing. In contrast a possible F# solution is (no you have not fallen asleep, we have still to cover basic syntax):

```
let rec sum data =
  match data with
  | []      -> 0
  | x :: xs -> x + sum xs
```

In this case the stack keeps track of our position in **`data`** and none of the variables change. It is also worth noting that there is no explicit type information in sight, the compiler can infer it all (how is another story).

What the above fails to highlight is that functional programming permits/encourages functions as parameters to functions, higher-order functions, which in turn leads to expressive elegance.

Let me inject a personal note at this stage. I first learnt BASIC at school (yes, I know Dijkstra’s views) and then the usual suspects in the ALGOL tradition with passing nods at a few others along the way. I have great difficulty giving up mutable variables altogether and the idea that *all code* should be side effect free is, to put it mildly, counter intuitive. The world has state, and that state changes. Input/output is a pretty big side effect. In an effort to get around side effects, computer scientists came up with monads, and with much syntactic sugar I can almost swallow one. Fortunately, F# hides many of its monads (called workflows in F#), permits mutable variables and supports a functional programming style. At last a functional language I can use that has the same general performance characteristics as other imperative languages.

JOE WOOD

Joe Wood wrote his first computer program almost 35 years ago, and sepnt his career developing software for a variety of large real-time systems. One day he might know how to write software, but strongly suspects that people problems trump technical issues all the time. He can be contacted at jawood@iee.org

Hunting the Snark (Part 5) (continued)

genuine discussion, or we can abdicate and let others decide what we are to become in the future. ■

References

[0] http://download.boulder.ibm.com/ibmdl/pub/software/dw/rationaledge/dec00/FromCrafttoScienceDec00.pdf

[1] Actually, trial and error is not completely dead in engineering. Many years ago when I was studying physics and engineering at university, I used to travel home by train. The train went past a very large power station, and in a field next to it was a collection of unused cooling towers in various stages of collapse.

I later met an engineer who had worked on those towers. He explained that the equations governing cooling towers were too complex to be solved (this is in the days when computers were big and expensive), and so the companies who designed cooling towers had built a set of towers with successively thinner walls until they got to the stage where the tower collapsed, to find the limits empirically. Today, of course we would solve those equations on a computer, but even in those days they knew how to prove the towers were safe, even if they couldn’t solve the equations to actually prove they were safe.

[2] We are not alone! The social sciences have been undergoing a similar struggle to find a basis for their discipline for even longer.

It is worth mentioning that there is no hard and fast definition of pure functional programming, and F# is able to support a spectrum of functional programming styles; from ‘thou shalt not use any side effects’ to ‘whatever works for you’. In fact, F#’s internal version of **`sum`** uses mutable variables, no recursion, and would look at home in any object-oriented code.

Getting F#

All the examples in this article have been written using Visual Studio 2010 Beta 1 with F# (free trial). There is also the compatible May 2009 CTP F# for use with Visual Studio 2008. F# also runs under Linux using mono, but without IDE support.

Basic syntax

Having, hopefully, convinced you to look at F#, we had better cover some of the basic syntax. Let’s start with the ever popular Hello World:

```
printfn "Hello, world"
```

`printfn` invokes a basic print routine **`printf`**, which supports arguments like C’s **`printf`**, and adds a trailing newline. Unlike C, the arguments to **`printf`** are type checked by the compiler. **`"Hello, world"`** is a string constant.

Function definitions

More interesting is a function to say hello to an individual, e.g.:

```
let hello name = printfn "Hello, %s" name
```

This defines a function, **`hello`**, which takes a single string argument **`name`**. The function calls **`printfn`** with two arguments, a string constant and the function parameter name. Notice that there are no parentheses anywhere, and no commas between the arguments to **`printfn`**. To invoke **`hello`** we say for example:

```
hello "Luke"
```

Unfortunately, there is a catch. When you call .NET libraries you have to add the parentheses and the commas between arguments. Some functional languages do not require any parentheses with nested function calls, but F# does.

F# determines the signature of **`hello`** to be **`string → unit`**. The → identifies **`hello`** as a function. The overall signature tells us that **`hello`** is a function that takes a single **`string`** argument and does not return a result (**`unit`**), i.e. **`void`** in C. If we wrote **`hello 5`**, the compiler would complain that **`5`** (an **`int`**) is not a **`string`**.

If we tweak the definition of **`hello`**, by replacing the format parameter **`%s`** with **`%A`**, the signature of **`hello`** becomes **`'a → unit`**. The **`'a`** indicates a generic parameter. In this case we can happily say **`hello 5`**, and we will see **`5`** printed out, as expected.

Basic bindings

Introducing variables is just as simple:

```
let star_wars_hero = "Luke Skywalker"
```

binds the string **`"Luke Skywalker"`** to **`star_wars_hero`**. This binding is immutable. Mutable variables can be introduced using:

```
let mutable top_artist = "Britney Spears"
```

and updated by:

```
top_artist <- "Robbie Williams"
```

You cannot change the type of the variable once determined. The right hand side is an expression and can be arbitrarily complex.

The **`let`** statement can also be used to perform limited pattern matching. Full pattern matching is beyond the scope of this introductory article.

Compound data types

The simple data types in F# are based on underlying .NET types. Beyond the simple data types, F# has the usual compound types, e.g. arrays, records, lists, tuples and maps, and access to the .NET collection types. Not all compound F# types (e.g. lists) have an equivalent .NET type, because the type semantics are different. There are library functions to convert between different representations when necessary.

Lists are used extensively in functional languages. In F# lists are immutable. Like C++ and Java, F# lists are homogeneous. Lists are written in the form **`[1;2;3;4;5]`** or equivalently **`[1..5]`**. Every list (except the empty list) has a head (the top element) and a tail (another list) of all the other elements. A single element list **`[42]`** has a head of **`42`** and the tail is the empty list **`[]`**. If we want to add a new element to the front of a list, we can use **`42 :: alist`**. You can add a list to the end of a list using the syntax **`alist @ blist`**.

The following expressions are all equivalent:

```
[1..5]
1 :: [2..5]
1 :: 2 :: 3 :: 4 :: 5 :: []
[1..4] @ [5]
[1..3] @ [4;5]
```

`::` is basically just a shorthand for the **`List.Cons`** function, e.g.:
`hd :: tl` is equivalent to **`List.Cons (hd, tl)`**

In a similar way **`@`** is just a shorthand for **`List.append`**, e.g.:
`alist @ blist` is equivalent to **`List.append alist blist`**

However we cannot use **`List.Cons`** or **`List.append`** or **`@`** in pattern matching.

You can access the elements of a list by index notation, e.g. **`alist.[4]`**. The **`.`** in the previous expression was not a mistake, the F# team are hoping to resolve this unusual syntax at some future date. In general any class can have index properties which are accessed by the **`.property[nth]`** style syntax. A special case is the property called **`Item`** which does not need the symbol **`Item`** in the index notation.

`alist.[4]` is a shorthand for **`List.nth alist 4`**

All F# lists are internally stored as pairs, the actual list element and a pointer to the next element. Hence looking for an element other than the head means chasing down a list of pointers, and hence expressions involving **`@`** or **`.[nth]`** should be avoided as they are computational expensive.

Tuples are ordered (fixed size) sets of heterogeneous data-types. For example **`(101, "Dalmatians")`** is a 2-tuple (aka a pair) of type **`int * string`**. The only way to access the individual elements of a tuple is either by pattern matching or by specific library functions. For example:

```
let mktuple x y = (x, y)
let t = mktuple 101 "Dalmatians"
```

Then we can use pattern matching to access the elements of **`t`**, using:

```
let (a, b) = t
```

and hey presto we have **`a = 101`** and **`b = "Dalmatians"`**. In the special case of a pair the library functions **`fst`** and **`snd`** extract the first and second elements respectively.

Please be aware that a function which takes two arguments is completely different from a ‘similar’ function that takes a 2-tuple as the single argument, i.e.:

```
let add_2arg x y = x + y
// standard 2 argument function
```

has signature **`int → int → int`** , and

```
let add_2tuple (x, y) = x + y
// single 2-tuple function
```

has the signature int **`* int → int`**.

End of line comments are introduced by `//` and in-line comments are bracketed by `(*` and `*)`. XML documentation can be added by the special comment symbol `///`, however the details on using such documentation comments are still sketchy.

Lambda functions

In a functional language, Lambda (anonymous) functions are common. The lambda function to increment an integer is just:

```
(fun x -> x + 1)
```

So to increment 42, we could use the expression:

```
(fun x -> x + 1) 42
```

which would produce 43, as expected. In such a situation there would be little point in using a Lambda function.

List.map is a function that takes two arguments, viz.: firstly a function to be applied to each member of the second argument which is a List. For example to increment all the elements of the list **[1..5]**, we could write:

```
let incr x = x + 1
List.map incr [1..5]
```

Alternatively, using Lambda functions we could use:

```
List.map (fun x -> x+1) [1..5]
```

In functional programming there are many higher order functions like **List.map** that take a function as an argument. Lambda functions provide a way to write 'short' anonymous functions as arguments to these higher order functions.

It is possible to never use a Lambda function in F#, and just define every function before its use. In practice, using Lambda functions is common in any functional language, it is part of the linguistic style. Behind the scenes most functional language compilers convert a 'normal' **let** function binding into a **let** value binding and an anonymous function, e.g.:

```
let incr x = x + 1
```

is compiled as

```
let incr = (fun x -> x + 1)
```

sum in detail

We have how covered sufficient syntax to take a more detailed look at **sum**. Recall that **sum** is defined by (the line numbers are for ease of reference):

```
let rec sum data =          // 1
  match data with           // 2
  | []      -> 0            // 3
  | x :: xs -> x + sum xs   // 4
```

The only new syntax on line 1 is **rec**, which declares **data** to be a recursive function. Indentation is important, as it denotes scope (like python).

match (line 2) starts a pattern match expression, matching data against the empty list (**[]**) or a non-empty list (**x::xs**). If the list is non-empty, then the head is bound to the local variable **x** and the tail is bound to the local variable **xs**.

Line 3 is the base case of the recursion and simply says that an empty list has a sum of 0.

Line 4 handles the general recursive case and says that the sum of a non empty list is the contents of the head (**x**) plus the sum of the tail of the list (**xs**).

You will notice that **sum** has no **return** statement, and has two exit points, i.e. lines 3 and 4. Multiple exit points are common in functional languages. Its one good reason for keeping functions short. There is a **return** statement in F#, but it is used only in asynchronous workflows.

A great beauty of such recursive definitions is that they are often 'obviously' true. Familiarity with mathematical induction is beneficial in seeing why.

We could rewrite the function's body (lines 2–4) using an if-then statement:

```
if data = [] then                     // 5
    0                                 // 6
else                                  // 7
    data.Head + sum data.Tail         // 8
```

Line 8 could be replaced by the following two lines:

```
let hd::tl = data     // 9, compiler warning
hd + sum2 tl          // 10
```

The binding in line 9 uses pattern matching to assign values to **hd** and **tl**, so that **hd** is the head of a list and **tl** the tail such that **hd::tl** equals data. This produces a compiler warning because the pattern match is not exhaustive, i.e. **data** could be empty.

Which version (recursive or iterative) and sub-version (pattern matching or if-then) of **sum** you prefer is, in most practical situations, a matter of personal taste. The recursive version could cause stack overflow for smaller sizes of **data** then the iterative version. However, this is unlikely to be problem in normal usage, and the use of an auxiliary function generally works for larger data sets. For example:

```
let sum data =
  let rec loop acc data =
    match data with
    | []      -> acc
    | x :: xs -> loop (acc + x) xs
  loop 0 data
```

Will not cause a stack overflow because of tail call optimisation, i.e. instead of pushing successive call frames onto the stack, a direct jump is made.

Of course in practice we would use **List.sum** to compute the sum of a list.

Informal testing would suggest that the recursive pattern matching version is the fastest. Using an if-then statement is about four times slower. Surprisingly, **List.sum** was slower than the fastest recursive version. Please bear in mind that this is a beta version of F#, and all the normal caveats on testing apply. The tests consisted of summing a list of 6,000 numbers, repeated 10,000 times and taking the average (max and min discarded) of 10 runs (see Figure 1).

Function	Time (ms)
List.sum	4,090
Recursive pattern matching	2,264
Recursive if-then test	9,555
Recursive if-then test with embedded let	8,100

Figure 1

Some real F# code

We looked at a few snippets of F#, but it is high time we looked at a longer example. In the September issue of CVu [3], Roger set puzzle 59, concerning decimal to hexadecimal (hex) converter. I submitted a critique [4] for this code. However, I though it was worth looking at an F# version. For space reasons you'll have to look back at your copy of November's CVu for the full C++ code.

The basic problem was to convert an unsigned (decimal) integer into its equivalent hex string. It is apparent that converting a decimal integer into a hex string is a special case of converting a decimal integer into a based string. So we shall start there (see Listing 1).

Now we can use partial application (creating a 'new' function from an existing function by 'freezing' one or more arguments) to quickly write **decToHex**:

```
// Given the input number, num, return its
// hexadecimal string representation
let decToHex = decToBase 16
```

That's the basic code for the problem. We still need a main program and a test harness.

Let's start by defining some test data. This is just a simple map (aka hash-table or dictionary) – see Listing 2.

Define **basic_tests** as a function to test our code above. **map** is a generic function found in most collections that applies the given function

Listing 1

```
let decToBase radix num =
  // decToBase is a function which takes two arguments, radix and num. Cannot use base as the name of
  // the first parameter as it is a reserved word in F#, so use radix instead
  let symbols = "0123456789abcdef" // symbols is the table for mapping decimal digits to characters

  // Test if radix is within the valid range, note we can use “||” in place of “or” but the
  // latter improves readability
  if (radix < 2) or (radix > 16) then
    // If invalid condition just raise an exception sprintf just prints into a string
    invalidArg "radix" ( sprintf "Illegal value '%i' for radix" radix )
    // This is roughly the equivalent of raise (new System.ArgumentOutOfRangeException(
    // "Illegal radix value" ))

    // Test that num is non-negative
    if num < 0 then
      invalidArg "num" "Input values must be non-negative"

    // Now we create an inner function, getBaseDigits, that converts the input number, n, into a list
    // of characters. Note: Must test for zero before calling this function. Note: This function
    // prepends the latest char onto the list and therefore the final list is in the correct order
    // Loop over the num and find the remainder when we divide by radix and convert the remainder to
    // base radix. Then we divide the current value of num by radix. Stop when we get to zero.
    let rec getBaseDigits str n =
      if n > 0 then
        let c = symbols.[ n % radix ]
        getBaseDigits (c :: str) (n / radix)
      else
        str

    // Convert num to list of radix digits, note the check for 0 to satisfy getBaseDigits
    let digitList n =
      match n with
      | 0 -> ['0']
      | _ -> getBaseDigits [] n

    // and finally do all the work and convert to a string
    new string ( List.to_array ( digitList n ) )
```

to each element of the collection and returns another collection with one element for each element in the original collection. For example:

```
List.map (fun x -> x*x) [1..5]
```

returns **[2;4;9;16;25]**, i.e. each element incremented by one. **fold** is like **map**, but it threads a value through the calculation on each element of the collection returning the final value. For example:

```
List.fold (fun state x -> state+x) 0 [1..5]
```

returns **15**, i.e. the sum of the numbers **1..5**. We use **Map.fold** to keep a cumulate check on the state of our tests. We test each element of **mk_results** in turn, with an initial state of **true** (passed) – see Listing 3.

Now we need a function to get an integer from the user to allow for other inputs not tested in **basic_tests**. This is declared as recursive because if we do not get an integer we will call the function again. This is a standard functional programming while loop. We must tell the compiler that prompt is a string, so that it can resolve the overloaded **Write** function (Listing 4).

Finally we come to the main routine (Listing 5), invoked by running:

```
main ()
```

Listing 2

```
let mk_results =
  Map [ (0,  "0");   (1,  "1");
    (2,  "2");   (3,  "3");
    (4,  "4");   (5,  "5");
    (6,  "6");   (7,  "7");
    (8,  "8");   (9,  "9");
    (10, "a");   (11, "b");
    (12, "c");   (13, "d");
    (14, "e");   (15, "f");
    (16, "10");  (511,"1ff");
    (512,"200"); (513,"201");
    (522,"20a");
    (System.Int32.MaxValue, "7fffffff");
  ]
```

Listing 3

```
let basic_tests () =
  let passed =
    Map.fold (
      // Loop over all the expected results and
      // check that we get no errors, keep a
      // cumulative pass/fail flag
      fun state num expected ->
        // Evaluate current number
        let result = decToHex num
        let pass = (result = expected)
        if not pass then
          printfn "Warning: Test failed,
             decToHex(%i) yielded %s expected %s"
            num result expected
         // return cumulative result
          pass && state
      ) true mk_results
  // Print overall test result
  if passed then
    printfn "Good basic tests passed, over to
you"
  else
    printfn "Warning: At least one basic test
failed"
```

Parallel computation

That has been a rather quick sprint over some F# code. However, you might think – not unreasonably – 'interesting, but it's just another language'. To some extent it is the functional mindset behind the code that is more interesting than the final code, and we cannot capture that in print.

Let's try to up the interest a little. Recall **map** and **fold** in **basic_tests**, which sequentially processed a collection. We can write a parallel extension to map as follows (based on [5])

```
module Map =
  let pmap f (m:Map<'Key,'T>) =
    seq { for a in m ->
      async { return (a.Key, f a.Key a.Value) } }
    |> Async.Parallel
    |> Async.RunSynchronously
    |> Map.of_array
```

The **|>** (pipeline symbol) takes the result from the previous function's execution and piplines it to the next function.

We can then observe that in **basic_tests**, the main **fold** could be broken into two parts, a parallel run of each individual test and then a 'collect the result' phase. The parallel **map** phase now becomes:

```
let ptest results =
  Map.pmap (fun num expected ->
    let result = decToHex num
    let pass = (result = expected)
    // return a triple
    (pass, expected, result)
    ) results
```

The sequential **fold** part becomes:

```
let fold results =
  Map.fold (fun state num v ->
    // decode the result from the pmap anonymous
    // function back into a triple
    let (pass,expected,result) = v
    if not pass then
      printfn "Warning: Test failed, decToHex(%i)
         yielded %s    expected %s"
        num result expected
    state && pass
    ) true results
```

You may be concerned that there is no obvious type information about the triple beyond structural equivalence. This stems from C and C#'s lack of name equivalence. We could create a class to handle this, but I'll leave that as an exercise for the reader to think about.

We can then rewrite **basic_tests** as:

```
let basic_tests2 () =
  let passed =
    mk_results |> Map.pmap |> fold
if passed then
  printfn "Good basic tests passed, over to you"
else
  printfn "Warning: At least one basic test
     failed"
```

Of course, in this simple case it is not worth doing. But one can easily imagine that with computationally expensive tests on a multi-core machine it would be a great speed up. Please bear in mind that there is always some overhead in splitting a problem up and re-combining the results. When in doubt do your own speed tests. However, not everybody has the same kit, so what may be sensible on a brand new turbo charged multi-core beast might be best avoided on an old uni-core system. The good news is that **pmap** is built on top of .NET libraries which are intended to make the best use of available resources.

```
let rec getInteger ( prompt : string ) =
  System.Console.Write ( prompt )
  let str = System.Console.ReadLine ()
  // Try to convert input string into an integer.
  // Note: Use of two variables and that the
  // output from TryParse is placed in result
  // without requiring any mutable variables or
  // refs.
  let success, result = System.Int32.TryParse (
     str )
  if not success then
    // Conversion failure, prompt and try again
    System.Console.WriteLine ( "You must enter a
       number, please try again, -1 to stop" )
    getInteger prompt
  else
    // All OK just return the result
    result
```

Listing 4

```
let main =
  basic_tests ()
  // test it We must put the () after mainloop to
  // ensure that it is treated as a function
  let rec mainLoop () =
    let dec_int =
      getInteger "Please enter a decimal number "
    if dec_int >= 0 then
      printfn "%i in hexadecimal is %s" dec_int
         (decToHex dec_int)
      mainLoop ()
  mainLoop
```

Listing 5

Summary and conclusion

F# is a usable functional programming language with support for OO and imperative style programming. A new language will not solve your software problems, but different styles of programming will make you look at problems in new ways and that will give you new insights into your problems. ■

Acknowledgement

I am most grateful to Steve Love for proof reading the original and asking questions which made the article more comprehensible.

References

[1] Syme, D., Granicz, A., Cisternino, A. (2007). *Expert F#*, Apress

[2] pop-11. (2009). In *Wikipedia, The Free Encyclopedia*. Retrieved 12:08, October 16, 2009, from http://en.wikipedia.org/w/index.php?title=POP-11&oldid=312842936

[3] Orr, R., et al (2009) 'Code Critique Competition 59', *CVu*, 21.4

[4] Orr, R., et al (2009) 'Code Critique Competition 60', *CVu*, 21.5

[5] F_Sharp_Programming. (2009). In Wikibooks. Retrieved 12:28, October 16, 2009, from http://en.wikibooks.org/wiki/F_Sharp_Programming

Further reading

Below are a few web sites (in no particular order) that might prove useful.

http://msdn.microsoft.com/en-us/fsharp/default.aspx. Microsoft F# Developer Centre (official) web site.

http://cs.hubfs.net. Main community centre for F# developers, frequented by some of Microsoft's F# team.

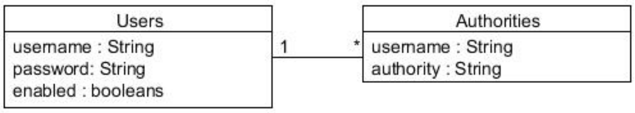
Data Access Layer Design for Java Enterprise Applications

Paul Grenyer explores a more object-oriented way of working with databases.

Java Database Connectivity (JDBC) can used to persist Java objects to databases. However JDBC is verbose and difficult to use cleanly and therefore is not really suitable for enterprise scale applications. In this article I will demonstrate how to replace JDBC persistence code with an Object Resource Mapper to reduce its verbosity and complexity and then, through the use of the appropriate patterns, show how you might design a more complete data access layer for a Java enterprise application.

The domain model

In order to demonstrate the design of an Enterprise Application Data Access Layer I am going to develop a solution to manage users and their roles (authorities) for the Spring Security database tables as described in Spring in Action [1]:



Spring Security expects two tables. The **`Users`** table holds a list of users consisting of a user name, password and flag to indicate whether or not the user is enabled. It has a one-to-many relationship with the **`Authorities`** table which holds the list of roles, stored as strings, each user has. Although Spring Security does not specify this to be enforced, it is sensible to use the **`username`** column as a primary key in the **`Users`** table and as a foreign key in the **`Authorities`** table (see Listing 1).

A Java abstraction of a Spring Security user might look something like Listing 2.

A **`User`** object has a user name, password, enabled flag and a list of roles. It has a constructor which initialises all of the fields, bar the roles, a method for adding individual roles and the appropriate getter for all fields.

Listing 1

```
CREATE TABLE [dbo].[Users]
(
  [username] [varchar](50) NOT NULL UNIQUE,
  [password] [varchar](50) ,
  [enabled] [bit] NOT NULL,
  CONSTRAINT Pk_Users PRIMARY KEY CLUSTERED
     ([username] ASC)
  ON [PRIMARY];

CREATE TABLE [dbo].[Authorities]
(
  [username] [varchar](50) NOT NULL,
  [authority] [varchar](50) NOT NULL,
  CONSTRAINT Pk_Authorities
  PRIMARY KEY CLUSTERED ([username],
     [authority] ASC),
  CONSTRAINT Fk_Authorities_User FOREIGN KEY
     ([username])
  REFERENCES [Users] ([username]),
  ON [PRIMARY];
```

The **`User`** class is the main object in our domain model.

Persisting objects with JDBC

Once a User **`object`** is instantiated with a user name, password, enabled status and a list of roles it is relatively straight forward, although verbose, to persist the object using JDBC (see Listing 3).

The save method makes a few assumptions:

1. The JDBC **`Connection`** object, **`con`**, is initialised before and cleaned up after the method call by other code.

Listing 2

```
public class User
{
  private String username;
  private String password;
  private boolean enabled;
  private List<String> auths
     = new ArrayList<String>();

  public User(String username, String password,
     boolean enabled)
  {
    this.username = username;
    this.password = password;
    this.enabled = enabled;
  }
  public void addAuth(String auth)
  {
    auths.add(auth);
  }
  public String getUsername()
  {
    return username;
  }
  public String getPassword()
  {
    return password;
  }
  public boolean isEnabled()
  {
    return enabled;
  }
  public List<String> getAuths()
  {
    return auths;
  }
}
```

PAUL GRENYER

An active ACCU member since 2000, Paul is the founder of the Mentored Developers. Having worked in industries as diverse as direct mail, mobile phones and finance, Paul now works for a small company in Norwich writing Java. He can be contacted at paul.grenyer@gmail.com

Listing 3

```
public static void save(
   User user, Connection con) throws SQLException
{
  try
  {
    con.setAutoCommit(false);
    final PreparedStatement userDetails
       = con.prepareStatement(
       "{call [dbo].spSaveUser(?,?,?)}");
    final PreparedStatement deleteAuthorities
       = con.prepareStatement(
       "{call [dbo].spDeleteAuthorities(?)}");
    final PreparedStatement saveAuthority
       = con.prepareStatement(
       "{call [dbo].spSaveAuthority(?,?)}");
    userDetails.setString(1, user.getUsername());
    userDetails.setString(2, user.getPassword());
    userDetails.setBoolean(3, user.isEnabled());
    userDetails.execute();
    deleteAuthorities.setString(
       1, user.getUsername());
    deleteAuthorities.execute();
    saveAuthority.setString(1,
       user.getUsername());

    for(String auth : user.getAuths())
    {
      saveAuthority.setString(2,auth);
      saveAuthority.execute();
    }
    con.commit();
  }
  finally
  {
    con.setAutoCommit(true);
  }
}
```

Listing 4

```
CREATE PROC [dbo].spSaveUser
   (@username [varchar](50),
    @password [varchar](50),
    @enabled [bit] )
AS
IF EXISTS(SELECT [username] FROM [dbo].[Users]
   WHERE [username] = @username )
UPDATE [dbo].[Users] SET [password] = @password,
   [enabled] = @enabled
    WHERE [username] = @username
ELSE
  INSERT INTO [dbo].[Users] (
     [username],[password],[enabled])
VALUES(@username,@password,@enabled)

CREATE PROC [dbo].spDeleteAuthorities
  ( @username varchar(50))
AS
  DELETE FROM Authorities WHERE [username]
     = @username;

CREATE PROC [dbo].spSaveAuthority
  ( @username varchar(50),
    @authority varchar(50))
AS
  INSERT INTO Authorities
([username],[authority])
  VALUES (@username,@authority);
```

2. The **PreparedStatment** objects can wait around to be cleaned up when the **con** object is cleaned up.
3. The stored procedures **spSaveUser**, **spDeleteAuthorities** and **spSaveAuthority** (see Listing 4) exist in the default database specified when the **con** object is initialised.

For more details on JDBC resource handling see 'Boiler Plating Database Resource Cleanup (Part I)' [2].

The **save** method takes the JDBC connection out of automatic commit mode, so that all the database operations occur within a transaction. The **finally** block at the end ensures that the connection is put back into automatic commit mode regardless of whether the operations succeeded or not. This, as I'll explain in the moment, is because the **User** table and the **Authorities** tables are updated separately and the changes should only be committed if both updates are successful.

The **User** table is updated first by getting the user name, password and enabled status from the **User** object and passing them to the **spSaveUser** stored procedure. The **spDeleteAuthorities** stored procedure is then used to remove all of the existing roles for the user from the **Authorities** table and finally the **spSaveAuthority** stored procedure is used in a loop to write the new roles to the table.

The **load** method, which is only slightly less verbose than the **save** method, is used to instantiate a **User** object from the database (Listing 5).

This **load** method makes the same assumptions as the **save** method, plus the existence of the **spGetUser** and **spGetAuthorities** stored procedures. There is no need for any transaction handling as the database is only being read.

The **spGetUser** stored procedure is used to get the user name, password and enabled status from the **Users** table if an entry for the specified user name exists. If it does then the **spGetAuthorities** stored procedure is used to get the user's roles and insert them into the **User** object.

On the surface the **save** and **load** methods look like simple, serviceable JDBC code, but in reality they are unnecessarily verbose and potentially difficult to maintain. An alternative is to use an Object Resource Mapper (ORM).

Object resource mapper

Using an ORM, such as Hibernate [3], can greatly reduce the amount of persistence code required. For example the **save** method could be reduced to Listing 6.

Listing 5

```
public static User load(String username,
   Connection con) throws SQLException
{
  final PreparedStatement getUser
     = con.prepareStatement(
     "{call [dbo].spGetUser(?)}");
  final PreparedStatement getAuthorities
     = con.prepareStatement(
     "{call [dbo].spGetAuthorities(?)}");
  getUser.setString(1, username);
  getAuthorities.setString(1, username);
  ResultSet rs = getUser.executeQuery();
  User user = null;
  if (rs.next())
  {
    user = new User(rs.getString(1),
       rs.getString(2), rs.getBoolean(3));
    rs = getAuthorities.executeQuery();
    while(rs.next())
    {
      user.addAuth(rs.getString(1));
    }
  }
  return user;
}
```

Listing 6

```
public static void save(User user,
   SessionFactory sessions)
{
  final Session session = sessions.openSession();
  final Transaction tx
     = session.beginTransaction();
  try
  {
    session.saveOrUpdate(user);
    tx.commit();
  }
  finally
  {
    session.close();
  }
}
```

Listing 7

```
public static User load(String username,
   SessionFactory sessions)
{
  final Session session = sessions.openSession();
  final Transaction tx
     = session.beginTransaction();
  User user = null;
  try
  {
    user = (User) session.get(
       User.class, username);
    tx.commit();
  }
  finally
  {
    session.close();
  }
  return user;
}
```

Of course there is slightly more to it than I have shown here, but I'll get to that shortly. The **SessionFactory** object passed into the method, among other things, manages connections to the database, which are served up as **Session** objects via the **openSession** method. The Hibernate notion of a session is somewhere between a connection and a transaction. Sessions must be closed regardless of success or failure and the most sensible place to do this is in a **finally** block. A transaction is started by calling **beginTransaction** on a **Session** object and committed by calling **commit** on the returned **Transaction** object. An object is persisted to the database by passing it to the **saveOrUpdate** method. If a row in the database with the same user name as the **User** object already exists it is updated, if one does not exist it is created.

As you can see there is no need to update the user and their roles separately, Hibernate takes care of all of that for you. There is also no need to write SQL or call stored procedures unless you want to, Hibernate does all that for you too. We'll see how after we've looked at the **load** method (Listing 7).

The **load** method works in much the same way as the **save** method, except instead of calling **saveOrUpdate** to persist an object it calls **get** to retrieve an object. The **get** method needs to know the type of the object it is retrieving and the object;s ID, in this case **username**. The returned object is then cast to the correct type (hopefully generics will appear in a later version and there'll be no need for the cast).

All of this relies on a properly configured **SessionFactory** object. Hibernate uses its own **Configuration** object which itself can be configured in lots of different ways, to create **SessionFactory** objects. The most straight forward way to configure it is with a **hibernate.properties** file and a Hibernate XML mapping file:

```
final SessionFactory sessions = new Configuration()
   .addClass(User.class)
   .buildSessionFactory();
```

Unless specified otherwise the configuration object looks for the **hibernate.properties** file in the classpath. A basic **hibernate.properties** file for a Microsoft SQL Server database looks something like Listing 8.

The connection url, username, password and driver are all self explanatory and the same as used by JDBC. The setting that is new is the dialect. Hibernate needs to know what sort of database it is connecting to so that it can generate the appropriate SQL and take advantage of any customisations. The dialect is set by specifying one of a number of different dialect objects supplied by Hibernate. It is also possible to write custom dialect objects for any database not supported.

The **Configuration** object also needs to know about the classes you want to persist to the database. Again there are lots of different ways to do this, but the simplest is to have a Hibernate XML Mapping file for each class along side it in the package. Using the **addClass** method to tell the **Configuration** object about the **User** class tells it to look for `User.hbm.xml` in `classpath:/uk/co/marauder/dataaccesslayer/model`. `User.hbm.xml` looks like Listing 9.

Hibernate XML mapping files are very easy to understand. As you can see the Java User **class** is mapped to the **Users** database table. All Hibernate persistable objects need an ID, which is specified by the **ID** tag. In this case **username** is used as the ID and is mapped to the **username** column in the **Users** table. Hibernate will use the **name** attribute to work out what the getter on the **User** object is called, in this case **getUsername**. The **generator** tag is used to specify if and how Hibernate should generate the ID for objects being saved for the first time. In this case we want to specify the ID ourselves, so the generator type is **assigned** and the ID

Listing 8

```
hibernate.connection.url=jdbc:sqlserver://localhost;DatabaseName=DataAccessLayer
hibernate.connection.username=user
hibernate.connection.password=secret
hibernate.connection.driver_class=com.microsoft.sqlserver.jdbc.SQLServerDriver
hibernate.dialect = org.hibernate.dialect.SQLServerDialect
```

Listing 9

```
<?xml version="1.0"?>
<!DOCTYPE
  hibernate-mapping PUBLIC
  "-//Hibernate/Hibernate Mapping DTD//EN"
  "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
  <class name="uk.co.marauder.dataaccesslayer.model.User" table="Users">
    <id name="username" column="username">
      <generator class="assigned"/>
    </id>
    <property name = "password" column = "password"/>
    <property name = "enabled" column = "enabled"/>
    <bag name="auths" table="Authorities" lazy="false">
      <key column="username" />
      <element type="java.lang.String" column="authority"/>
    </bag>
  </class>
</hibernate-mapping>
```

is given to the **User** object before it is persisted. The **property** tag is used to map the other fields to table columns. The **bag** tag tells Hibernate that the **User** class has a list of strings that should be written to the **authority** column in the **Authorities** table and that the foreign key relating them to the **User** object is **username**.

Obviously this is quite a simple mapping. Hibernate is capable or much more complicated object relationships including inheritance. Hibernate XML mapping files are not the only way to tell Hibernate about the classes you want to persist. Hibernate also provides some annotations, but I prefer to keep domain objects decoupled from the persistence mechanism as much as possible.

We're almost there. Unfortunately Hibernate needs a default constructor and won't use **User**'s current constructor. When loading objects it default constructs them and uses setters to initialise them. **User** doesn't have setters so they need to be added (Listing 10).

Hibernate is quite clever, it can find and call the constructors and methods it needs by reflection, even if they are private. So if it is undesirable for your domain objects to have a default constructor or particular methods that are needed by Hibernate they can be made private.

This is all that is needed for a fairly significant reduction in the amount of code that is needed. It also reduces the the amount of exception handling. However, it is possible to reduce it even further using the Spring Framework's [4] **HibernateTemplate** class (Listing 11).

HibernateTemplate takes care of getting and releasing Hibernate sessions and some of the transaction handling. It would normally be initialised with a **DataSource** in the Spring runtime's application context, but if you're not using Spring's application context it can be initialised with a **SessionFactory**:

```
final SessionFactory sessions = new Configuration()
  .addClass(User.class)
  .buildSessionFactory();
final HibernateTemplate hibernateTemplate
   = new HibernateTemplate(sessions);
```

Listing 10

```
public class User
{
  private String username;
  private String password;
  private boolean enabled;
  private List<String> auths
     = new ArrayList<String>();
  @SuppressWarnings("unused")
  private User()
  {}
  ...
  @SuppressWarnings("unused")
  private void setUsername(String username)
  {
    this.username = username;
  }
  @SuppressWarnings("unused")
  private void setPassword(String password)
  {
    this.password = password;
  }
  @SuppressWarnings("unused")
  private void setEnabled(boolean enabled)
  {
    this.enabled = enabled;
  }
  @SuppressWarnings("unused")
  private void setAuths(List<String> auths)
  {
    this.auths = auths;
  }
}
```

Listing 11

```
public static void save( User user,
   HibernateTemplate hibernateTemplate)
{
  hibernateTemplate.saveOrUpdate(user);
}
public static User load(String username,
   HibernateTemplate hibernateTemplate)
{
  return (User) hibernateTemplate.get(
     User.class, username);
}
```

I have demonstrated how the **load** and **save** methods for the **User** object can be reduced from several lines of code for a JDBC solution, down to a single line each with an ORM solution. Even with the Hibernate XML mapping file this is a significant reduction in code and complexity. However, this is not enough for a real Enterprise Application Data Access Layer.

Data mappers, registries and managers

The **load** and **save** methods separate the concern of persisting a **User** object from the object itself, and that is a good thing. However the user of the methods still knows they are persisting to a database and the mechanism used to persist to the database as they have to provide the appropriate **Connection**, **SessionFactory** or **HibernateTemplate** object. Worse still, when writing a unit test that wants to simulate persisting data to the database those very same **Connection**, **SessionFactory** or **HibernateTemplate** objects must be stubbed out. This is not a trivial exercise as each has many methods and acts as a factory for other objects that must also be stubbed.

Fortunately Martin Fowler solves this problem with two patterns from his *Patterns of Enterprise Application Architecture* [5] book. The DATA MAPPER, which moves data between objects and the database while keeping them independent of each other and the mapper itself and the REGISTRY, which is a well known object that other objects can use to find common objects and services.

The Data Mapper is an object that you can ask to load, save or perform some other sort of persistence operation on an object. In practice creating a Data Mapper for the **User** class is slightly more than refactoring the **load** and **save** methods into an class (Listing 12).

The **UserHibernateDataMapper** class is specific to **HibernateTemplate**, but similar classes could be written for a straight forward JDBC **Connection** or for a Hibernate **SessionFactory**. Once created and initialised with a **HibernateTemplate** the object can be passed around as needed and used to persist User objects without the user being aware of the implementation.

Listing 12

```
public class UserHibernateDataMapper
{
  private final HibernateTemplate
     hibernateTemplate;
  public UserHibernateDataMapper(
     HibernateTemplate hibernateTemplate)
  {
    this.hibernateTemplate = hibernateTemplate;
  }
  public void save(User user)
  {
    hibernateTemplate.saveOrUpdate(user);
  }
  public User load(String username)
  {
    return (User) hibernateTemplate.get(
       User.class, username);
  }
}
```

Listing 13

```
public class UserHibernateDataMapper extends
   HibernateDaoSupport
{
  public UserHibernateDataMapper(
     HibernateTemplate hibernateTemplate)
  {
    setHibernateTemplate(hibernateTemplate);
  }
  public void save(User user)
  {
    getHibernateTemplate().saveOrUpdate(user);
  }
  public User load(String username)
  {
    return (User) getHibernateTemplate().get(
       User.class, username);
  }
}
```

Spring also provides a helper class for Data Mappers called `HibernateDaoSupport` which, among other things, provides accessors for the `HibernateTemplate` (Listing 13).

For unit testing, data mappers that talk directly to a database need to be easily interchangeable with the equivalent mock objects. Mock objects are simulated objects that mimic the behaviour of real objects.

Listing 14

```
public interface UserDataMapper
{
  void save(User user);
  User load(String username);
}
public class UserHibernateDataMapper
   extends HibernateDaoSupport
   implements UserDataMapper
{
  public UserHibernateDataMapper(
     HibernateTemplate hibernateTemplate)
  {
    setHibernateTemplate(hibernateTemplate);
  }
  @Override
  public void save(User user)
  {
    getHibernateTemplate().saveOrUpdate(user);
  }
  @Override
  public User load(String username)
  {
    return (User) getHibernateTemplate().get(
       User.class, username);
  }
}
public class MockUserDataMapper
   implements UserDataMapper
{
  private Map<String,User> users
     = new HashMap<String,User>();
  @Override
  public User load(String username)
  {
    return users.get(username);
  }
  @Override
  public void save(User user)
  {
    users.put(user.getUsername(), user);
  }
}
```

The easiest way to make the real data mappers and the mock data mappers interchangeable is for them both to implement the same interface. This is achieved by Extracting the Interface [6] of the real data mapper and then creating a mock implementation (Listing 14).

Of course you can have as many implementations of the interface as you want, so you could have a Hibernate implementation, a JDBC implementation and a mock implementation for use in different systems if you wanted too.

Each data mapper only needs to be created once. You could create and destroy them when they are needed, but then the underlying JDBC or Hibernate object would need to be passed around instead and the user would no longer be abstracted from the particular data access implementation being used. However, passing individual data mappers to everywhere they are needed can be tedious and it is much easier to pass around a single object that can be asked for the required data mapper.

Martin Fowler's Registry pattern describes one such object. Just to remind you, the `Registry` is a well known object that other objects can use to find common objects and services. Patterns describe solutions to problems, not implementations. Therefore the registry used to store and retrieve data mappers can be very different to the example Fowler suggests. The implementation I have chosen is this:

```
public interface DMRegistry
{
  <T> T get(Class<T> interfaceType,
     Class<?> object);
}
```

Having a registry interface makes the interchanging of different implementations of registries easier and the passing round of registries less coupled. For example you might have a JDBC data mapper registry, a

Listing 15

```
public abstract class AbstractDMRegistry
   implements DMRegistry
{
  private final Map<Class<?>, Object> map
     = new HashMap<Class<?>, Object>();
  public <T> void add(final Class<?> objectType,
     final T dataMapper)
  {
    map.put(objectType, dataMapper);
  }
  @Override
  public <T> T get(Class<T> interfaceType,
     Class<?> object)
  {
    final T dataMapper = interfaceType.cast(
       map.get(object));
    if(dataMapper == null)
    {
      throw new IllegalStateException( object
         " not found in registry "
         + getClass().getName());
    }
    return dataMapper;
  }
}
```

Hibernate data mapper registry and a mock data mapper registry. If your code using the data mappers takes a **`DMRegistry`** you can easily change the implementation just by passing the required registry to it. (Listing 15.)

`AbstractDMRegistry` is an abstract class that holds the common implementation for all data mapper registries. All data mappers are identified by the type of the object they map. The **`add`** method is used to map the object type to a data mapper instance. The **`get`** method takes the expected interface for the data mapper and the object type, looks it up and returns the appropriate data mapper. If a data mapper for the supplied object type is not present an exception is thrown.

Finally you need a specific implementation of the data mapper registry. A **`HibernateTemplate`** implementation might look like this:

```
public class HibernateTemplateDMRegistry
   extends AbstractDMRegistry
{
  public HibernateTemplateDMRegistry(
     HibernateTemplate hibernateTemplate)
  {
    add(User.class,new UserHibernateDataMapper(
       hibernateTemplate));
  }
}
```

The data mapper for the User object is created using a supplied **`HibernateTemplate`**, in the constructor and added to the registry. Any number of data mappers can be added to and accessed from the registry. Other implementations of a registry would be very similar, but instantiate different data mappers. A **`HibernateTemplateDMRegistry`** is instantiated like this:

```
final DMRegistry registry
   = new HibernateTemplateDMRegistry(
   hibernateTemplate);
```

and used like this:

```
final UserHibernateDataMapper mapper
   = registry.get( UserHibernateDataMapper.class,
   User.class);
mapper.save(user);
final User newUser
   = mapper.load(user.getUsername());
```

Once the data mapper has been retrieved from the registry it is simple to use, but the code to retrieve it is verbose and could end up being repeated throughout your code. One way to get around this is to use a Facade [7] to ‘manage’ the object being persisted (Listing 16).

Listing 16

```
public class UserManager
{
  private final DMRegistry registry;

  public UserManager(DMRegistry registry)
  {
    this.registry = registry;
  }
  private UserHibernateDataMapper getMapper()
  {
    return registry.get(
       UserHibernateDataMapper.class,User.class);
  }
  public void save(User user)
  {
    getMapper().save(user);
  }
  public User load(String username)
  {
    return getMapper().load(username);
  }
}
```

The **`UserManager`** uses a reference to data mapper registry to obtain the data mapper for the **`User`** object and uses it to save and load **`User`** objects. A **`UserManager`** can be passed around instead of a data mapper registry and reduce the verbosity and repetition of code without loosing any of the flexibility and does not care what sort of data mapper registry it has been passed. More operations can be added to managers easily and if an operation becomes more complicated and, for example, requires multiple data mapper calls the manager becomes the ideal place to add things like encompassing transactions.

Finally

I have shown how to reduce the verbosity and complexity of JDBC persistence code using an ORM. I have also shown how a data access layer could be written that is suitable for an enterprise application. It allows simple interchanging of different database persistence implementations, including a mock object implementation to aid automated unit testing.

By using the basic form of the Spring Security database I have demonstrated very simple use of an ORM and object managers. ORMs can be used to persist far more complicated object relationships and managers used to do far more. Although it is only the tip of the iceberg this article should give a firm grounding for more complex enterprise application data access layers. ■

References

[1] *Spring in Action* by Craig Walls, Ryan Breidenbach, Manning Publications; 2 edition ISBN: 978-1933988139

[2] ‘Boiler Plating Database Resource Cleanup – Part I’ by Paul Grenyer, http://www.marauder-consulting.co.uk/Boiler_Plating_Database_Resource_Cleanup_-_Part_I.pdf

[3] Hibernate: https://www.hibernate.org/

[4] Spring Framework: http://www.springsource.org/

[5] *Patterns of Enterprise Application Architecture* by Martin Fowler, Addison Wesley ISBN: 978-0321127426

[6] *Refactoring: Improving the Design of Existing Code* by Martin Fowler, Addison Wesley, ISBN: 978-0201485677

[7] *Design patterns : elements of reusable object-oriented software* by Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Addison Wesley, ISBN: 978-0201633610

Deciding Between IF and SWITCH When Writing Code (Part 2)

Derek Jones concludes his study of programmers' habits.

This article is the second of two that investigates the possible factors influencing developers in their choice of selection statement, i.e., deciding whether to use an if-statement or a switch-statement to implement some desired functionality. Two sources of data are analysed: the first article[2] analysed measurements of existing code and this second one discusses the results of an experiment carried out at the 2009 ACCU conference.

The source code measurements in the first article showed that the switch-statement rapidly becomes the selection statement of choice as the number conditionally executed statement sequences increases; see Figure 1. The 2009 ACCU experiment asked subjects to write various function definitions whose specification each involved behaviour that depended on one variable that could take on various values.

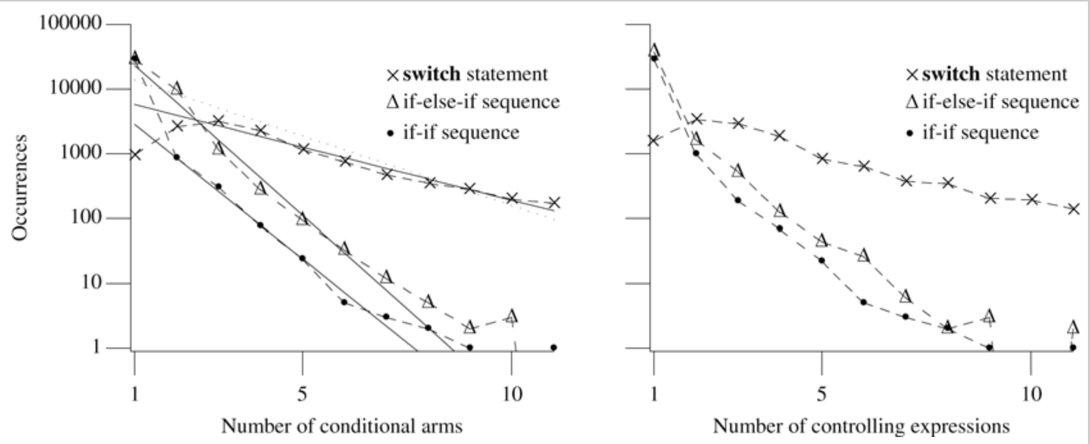


Figure 1

The left plot is of occurrences of if-else-if, if-if (uncertainty about which sequence a single if-statement, without an else-arm, belonged to was resolved by including it in both sequences) and switch-statements having the given number of conditional arms. The right plot is the number of controlling expressions in a if-else-if and if-if sequence and the number of case-labelled statement sequences (i.e., it excludes the effect of any default) in a switch-statement. The solid lines are a least-squares fit of the data to an exponential function. The dotted line is a fit to the sum of all sequences having a given number of arms.

Previous experience has shown that when asked to solve one simple problem developers often quickly fall into using a fixed pattern when answering. One of the aims of the ACCU experiments is for them to reflect actual work practices (to be ecologically valid is the technical terminology). In an attempt to prevent subjects following a pattern of answers that they would not follow in a work related environment each problem was split into two independent sub-problems.

The hypothesis

The experimental problem contained two independent subproblems and separate hypothesis derived for each subproblem.

1. Recognition of function call sequence order. Subjects are more likely to correctly recall the sequence of a previously seen sequence of function calls if the names of those functions follow a commonly occurring pattern, e.g., alphabetic/numeric order or having a (hopefully) recognizable order such as the sequence start-process-end. The details are discussed below.
2. `if`/`switch` statement choice. The decision on whether to use an `if` or `switch` statement is strongly affected by the number of conditional arms expected to appear in the final code. A secondary hypothesis is also tested:
   - Experienced developers will be aware that code written today frequently has to be modified in the near future because of changes to requirements or other parts of the code base. A developer's expectation of future changes to code that is being written now may also be a factor in the choice of selection statement.

     To test this hypothesis problems were designed to either involve quantities that subjects were thought likely to consider as being open-ended (living space related information was used e.g., trees in a garden) or to involve quantities thought likely to be considered to be closed by subjects (human body related information was used e.g., number of fingers on a hand). The details are discussed below.

The differences between the left and right plots in Figure 1 are caused by contributions from `else` and `default`. In the case of an if-if sequence the `else` can only appear on the final if-statement and a default-label cannot be mixed with case-labels on the same statement sequence. The solid lines are a least-squares fit of the data to an exponential function.

Experimental setup

The experiment was run by your author during a 40 minute lunch time session at the 2009 ACCU conference (www.accu.org) held in Oxford, UK; between 275 and 325 professional developers attend this conference every year. Subjects were given a brief introduction to the experiment, during which they filled in background information about themselves, and then spent 27 minutes answering problems. All subjects volunteered their time and were anonymous.

The problem format was very similar in form to several previous ACCU experiments [1].

The problem

Figure 2 is an excerpt of the text instructions given to subjects (your author went through these instructions and the associated example once everybody had settled down in the room, prior to them answering any problem).

DEREK JONES

Derek used to write compilers that translated what people wrote. These days he analyses code to try and work out what they intended to write. Derek can be contacted at derek@knosof.co.uk

What you have to do

This is not a race and there are no prizes for providing answers to all questions. Please work at a rate you might go at while reading source code. The task consists of remembering the sequence of three function calls and recalling this sequence later The function calls appear on one side of the sheet of paper and your response needs to be given on the other side of the same sheet of paper.

1. Read the function call sequence like you would when carefully reading lines of code in a function definition.
2. Turn the sheet of paper over. Please do **NOT** look at the function calls you have just read again, i.e., once a page has been turned it stays turned.
3. To create a time delay between reading the sequence of function calls and having to recall the sequence you are asked to write some code to assign a value to some variable. You do not know where these variables are defined, it may be in some other compilation unit, or locally within the current function.

   The function **`set_variable`** has one parameter and this can have any of the numeric values listed above the function skeleton, to the left. The value to be assigned to a particular variable appear to the right of the input value to which they apply.

   Only one function has to be written for each problem.

   The code can be written in a language of your choice (it would simplify subsequent analysis if either C, C++, Java, Pascal or C# were used).

   The values and variables either involve parts of the human body or relate to a person’s home.
4. Once you have written the code you are now asked to recall the sequence of function calls seen on the previous page.
   - if you remember the sequence circle the appropriate list,
   - if you feel that, in a real life code comprehension situation, you would reread the original function call sequence, circle the *refer back* column on the right.

If you do complete all the questions do **NOT** go back and correct any of your previous answers.

```
op_1();

op_2();

op_3();
```

```
 1 -> X = "Intel";
20 -> y = "Motorola";
33 -> W = "IBM";
41 -> p = "Sun";
```

Two of the ways in which the appropriate assignment might be performed

```
void set_variable(int company)
{


if (company == 1)
  X = "Intel";
else if (company == 20)
  y = "Motorola";
else if (company == 33)
  W = "IBM";
else if (company == 41)
p = "Sun";

}
```

```
void set_variable(int company)
{
  switch(company)
  {
    case 1: X = "Intel";
            break;
    case 20: y = "Motorola";
            break;
    case 33: W = "IBM";
            break;
    case 41: p = "Sun";
            break
  }
}
```

```
op_2();     op_3();     op_1();     op_2();
op_1();     op_1();     op_2();     op_3();      refer back
op_3();     op_2();     op_3();     op_1();
```

Figure 2

Background to problem generation

The problems and associated page layout were automatically generated using various awk scripts to generate troff, which in turn generated postscript. The source code of the scripts is available from the experiment’s web page.[3]

Sequence of function calls

The names of the functions used for each sequence of three function calls was generated as follows:

- A total of 30 different sets of three function names was created. In seventeen of these sets the names were English words having the property that it was possible to place the names in an ordered relationship that many English speakers were thought likely to recognise, e.g., toe, foot, leg. Four of the sets contained English words having no obvious ordering relationship between them (e.g., morning, collide, gutter), five of the sets contained unrelated nonword-like sequences of letters (e.g., cgy, pdl, kxr) and four of the sets contained unrelated word-like sequences of letters (e.g., esak, dard, sule). The complete list of names used is available from the experiment’s web page.[3]
- For each subject, the sequence of functional calls used in a problem was created by randomly selecting a previously unselected, set of three function names. The ordering of the names was randomised and this sequence was printed. The list of four possible answers was

generated by creating three other unique random orderings of the names, each differing from the printed sequence, and printing out the four possible answers in a randomly selected order (*refer back* was added as a fifth possible answer).

Specification of function definition

The coding part of the problem answer sheets seen by subjects had two parts:

- a list of value/assignment pairs. When the function parameter had a given value a specified assignment was required to be executed.

  The problem description did not specify any information on the kind of source statements to use.
- a function definition template within which enough white-space was provided for subjects to write their answer. The function took a single parameter whose name was intended to generate a particular semantic association in the subject's mind.

```
23 -> K=0;
46 -> R=1;
 4 -> N=2;

void set_variable(int num_ears_pierced)
{

// Sufficient vertical white space here to
// write code

}
```

It was decided that the number of conditions contained in each problem specification be either 3, 4 or 5. The detailed information present in Figure 1 was not available when this decision was made, otherwise problem specifications containing 2 conditions would also have been included.

Semantic information on the kind of operation being performed by the function was indicated via the name of the parameter variable being tested against and the numeric or string literal being assigned. For instance, the variable name num_garden_sheds is intended to convey to subjects that it holds a value representing a number of garden sheds (another possible interpretation is the number of garden sheds for sale or visible from some vantage point; both are open-ended in that the number of sheds is unlikely to be thought to be limited in real life to the range given in the problem), and requiring the assignment of one of the three strings 'two handed', 'one handed' and 'hands free' is intended to convey that the function involved humans using their hands (a closed set in the sense that humans have a maximum of two hands).

A total of 30 different possible variables and associated values were created. Half of these variables had names intended to indicate something involving the human body (the closed set) and the other half had names relating to human habitation (the open set). The ordering of the list of 30 possible variables was randomised for each subject.

The extent to which subjects considered it likely that the range of values held by a variable having a given name will change at a future date was found by asking them, at the end of the experiment: "For each of the problems you answered please specify what you think the likelihood is, on a scale of 1 to 10, that at some future date additional items will be added to the list of possible parameter values. 1 means extremely unlikely while 10 means inevitable." (the list of questions appeared in the same order as the list of problems they encountered).

The identifier used in each assignment statement was randomly chosen from a set of single letter identifier names and the order in which the assignment statements (for each problem) was listed was also randomized. The complete list of names and corresponding literal values used, along with subjects' evaluation of the likelihood of additional values being added is available from the experiment's web page. [3]

Threats to validity

For the results of this experiment to have some applicability to actual developer performance (i.e., to be ecological valid) it is important that subjects work through problems at a rate similar to that which they would process source code in a work environment. Subjects were told that they are not in a race and that they should work at the rate at which they would normally process code. However, developers are often competitive and experience from previous experiments has shown that some subjects ignore the work rate instruction and attempt to answer all of the problems in the time available. To deter such nonwork-like behaviour during this experiment the problem pack contained significantly more problems than subjects were likely to be able to answer in the available time (and this was pointed out to subjects during the introduction).

If subjects are asked to repeatedly write the same kind of coding construct situation they may not behave in the same way as when involved in having to use a variety of different constructs. In the ACCU experimental context it would not be practical to ask subjects to write lots of different kinds of code. Within the constraints of the ACCU experiment it was only practical to use two independent subproblems and it is hoped that this would be sufficient to prevent subjects rapidly falling into a nonwork-like fixed pattern of behaviour.

The structure of the problem used follows a pattern that is often encountered when trying to comprehend source code: see information (and remember some of it), perform some other task and then perform a task that requires making use of the previously seen information. In this experiment the three activities were: remember some coding information, write some unrelated code and finally recognize the previously remembered information.

It is possible that when answering a series of problems having the same overall structure subjects may decide to use the same coding technique for each problem, making their answers unrepresentative of what they would have written outside of the context of this experiment.

The **if**/**switch** problem involved a component that relied on subjects making a semantic association with the name of a variable and making use of that information. An experiment that uses semantics as the control variable depends on subjects recognizing the appropriate semantic content in the problem being answered. If the anticipated semantic effect does not appear in the results one explanation is that subjects failed to extract the implied semantic information, another is that subjects extracted different information from that intended by the experimenter; a subject's failure to make use of the intended semantic information is also an explanation.

Subject strategies and motivations

Talking with subjects who have taken part in previous ACCU experiments uncovered that they had used a variety of strategies to remember information in remember/recall problems and had problem completion motivations that had not been anticipated. The analysis of the threats to validity of these experiments[1] discussed the question of whether subjects traded off cognitive effort on one subproblem in order to perform better on another subproblem, or carried out some other conscious combination of effort allocation between subproblems. To learn about strategies used during this experiment, after 'time' was called on problem answering, subjects were asked to list any strategies they had used (two sheets inside the back page of the handout had been formatted for this purpose).

Results

It was hoped that at least 30 people (on the day 12) would volunteer to take part in the experiment and it was estimated that each subject would be able to answer 20 problem sets (on the day 16.6 sd 4.1; a total of 199 answers) in 20–30 minutes (on the day 27 minutes).

The average amount of time taken to answer a complete problem was 97.6 seconds. No information is available on the amount of time invested in trying to remember information, answering the coding subproblem, and then thinking about the answer to the sub-problem (i.e., the effort break down for individual components of the problem).

The average professional experience of the subjects was 12 years (standard deviation 7.4).

if/switch choice

All but one subject primarily always used either an if-else statement (2 subjects) or a switch-statement (9 subjects; a few subjects answered one question using an if-statement). One subject used an if-else statement when the specification contained three conditions and a switch-statement when the specification contained more than three conditions.

Based on the measurements used to plot Figure .1, with three conditional arms the probability of a switch-statement being used is 73% (the value with four conditional arms is 89%). The likelihood that 11 out of 12 subjects will primarily always use a switch-statement is very low.

The pattern of usage seen in the experiment answers would not generate the relative frequency of occurrences seen in the source code measurements. Possible reasons for this experimental behaviour include:

- The relative frequencies seen in Figure .1 are caused by something other than developers basing their choice of if/switch statement usage on the number of conditional arms.
- In an attempt to improve their performance in the remember/recognise subproblem subjects decided to always give either an if-statement or a switch-statement answer (i.e., they assumed that by not making a separate choice for each answer they were more likely to correctly answer the function sequence subproblem).

After time was called on answering problems subjects were asked to: 'Please list any strategies you used when writing the code'.

Strategies listed included: 'Always the same', 'Dumb way, because little info (followed example)', 'do not analyse context' and 'Minimise amount of writing'. The subject who used a mixed if/switch strategy wrote: '3 items or less -> if-else chain, more items -> switch case'.

Subjects used two orders for testing the parameter against the various numeric values listed in the problem specification:

- The two subjects who primarily used if-else tested the numeric literals in the order in which they appeared in the specification. The subject who only used if-else in the three conditional arm problems sorted the numeric values and tested them in this order.
- Seven of the nine subjects who primarily used switch tested the numeric literals in the order in which they appeared in the specification, the other two subjects plus the subject who used an if-else in the three conditional arm problems sorted the numeric values and tested them in this order.

Recognition performance

Subjects saw a sequence of three function calls and later had to either select one out of four sequences or select *refer back*. One of the four sequences was the same as that seen earlier, so there a random answer had a 25% chance of being correct.

Of the 199 problems answered by the 12 subjects 84.9% were correct, 12.6% were *refer back* and 2.5% incorrect (in all 5 incorrect cases the first function name in the answer sequence was correct).

All of the incorrect answers were given by just 25% of the subjects; this is not statistically significant because of the very small number of incorrect answers involved.

The *refer back* answer was not given by 58% of subjects, but was given quiet a few times (mean 31.5% of answers) by 33% of the subjects. There are several possible explanations for some subjects giving many *refer back* answers, including:

- Self-knowledge, or metacognition, is something that enables a person to evaluate the accuracy of the memories they have. Perhaps these subjects have poor metacognitive abilities (i.e., they underestimated the accuracy of their memories and might have been correct had they risked giving an answer),
- these subjects were very cautious, or risk averse, people,
- these subjects exhibited poor short term memory performance during the experiment (which may be due to being tired or having a low short term memory capacity).

After time was called on answering problems subjects were asked to: 'Please list any strategies you used to remember the sequence of items'

Strategies commonly listed by subjects for remembering a sequence of function definitions included: noting if the items were in alphabetic order, reverse alphabetic order, logical order and noticing when the initials formed an acronym they knew. These strategies implied that subjects were not remembering the names of the functions but some pattern that could be used to later recognise which sequence to circle (for their answer). This method of operation works because subjects were only asked to remember one sequence at a time, something that is not that common in a software development environment.

Conclusion

The results did not produce any evidence that significantly supported the hypothesis concerning developer choice of `if`/`switch` statements or that a known ordering relation between a sequence of functions names aided later recognition of that sequence.

Subject made so few mistakes in the function sequence recognition problem that it is not possible to reliably detect any patterns in the mistakes made. It is difficult to know how to structure a recognition problem that would generate a sufficient number of subject mistakes while maintaining a reasonable degree of ecological validity.

The memory problem could have been structured as a recall problem (i.e., ask subjects to write down the sequence of names without being given any external clues; when writing software developers have to recall the appropriate sequence to call functions and when reading existing source spot when a sequence of calls is incorrect). This experiment was not based on recall because it was thought, prior to the experiment, that this would require too much cognitive effort from subjects who might then give many *refer back* answers. All but one subject used the same coding construct for all `if`/`switch` problem answers. This behaviour may have been an artifact of the experimental situation. Future experiments investigating `if`/`switch` decision making might like to include problems containing two conditional arms. When describing the problem to subjects and telling them what they are being asked to do it might be worthwhile stressing that they should give the `if`/`switch` part of the experiment equal weighting and not use a coding strategy aimed at improving their performance on other parts of the problem. ■

Further reading

For a readable introduction to human memory see *Essentials of Human Memory* by Alan D. Baddeley; a more advanced introduction is given in *Learning and Memory* by John R. Anderson. An undergraduate level discussion of some of the techniques people use to solve everyday problems is provided by *Simple Heuristics That Make Us Smart* by Gerd Gigerenzer and Peter M. Todd. An excellent introduction to many of the cognitive issues that software developers encounter is given in *Thinking, Problem Solving, Cognition* by Richard E. Mayer.

Acknowledgements

The author wishes to thank everybody who volunteered their time to take part in the experiments and ACCU for making a slot available, in which to run the experiment, at the 2009 conference.

References

1 D. M. Jones 'Developer beliefs about binary operator precedence', *C Vu*, 18(4):14–21, Aug. 2006.

2 D. M. Jones 'Deciding between if and switch when writing code', *C Vu*, 21(5):14–20, Nov 2009.

3 D. M. Jones 'Experimental data and scripts for deciding between if and switch', http://www.knosof.co.uk/cbook/accu09.html, 2009

A Game of Skill

Baron Muncharris sets a challenge.

Sir R-----! It seems that Madame Fortune has once again willed that our paths cross in this fine hostelry. Come join me in raising a glass in honour of her divine wisdom.

Might you be willing to pit your wits and your purse against mine in a game of skill?

Excellent! I should not have expected a Gentleman of such high standing as your good self to shy from some sport.

I learnt the game banqueting in Valhalla after I was slain protecting the Empress of Russia from a particularly ardent suitor. Needless to say, the rogue fared worse than I, but that is a tale for another evening.

The game was invented by Odin as a peaceable means to settle disputes between Valkyries after an especially boisterous banquet in which his great hall was very nearly reduced to rubble during an argument over whether it was proper to serve mead or ale during the cheese course.

It has since become quite the popular after dinner entertainment amongst the more genteel guests at his banquets, as I'm sure you will appreciate once I explain to you its play.

Here, I have lain before you 9 cards

A♥ 2♥ 3♥ A♠ 2♠ 3♠ A♣ 2♣ 3♣

We shall take turns drawing cards from this stock and contrive to build a trick of three cards of the same suit or of the same face value.

You shall draw first and have the advantage of being allowed to take three cards that all differ in both suit and face value as a trick, provided that they include the very first card you draw.

For example, if the first card you draw is 2♣, then the following three tricks are amongst those that might win the game for you

A♥	2♣	3♠
A♠	2♠	3♠
A♥	A♣	A♠

If you can press your advantage and build a trick before I, you shall have the game and a coin from my purse. If however I build a trick first, or if the stock of cards is exhausted, then I shall have the game and likewise a coin of yours.

Upon hearing these rules, that disreputable student acquaintance of mine became somewhat agitated and started wittering on about tactics and the eyes of Morpheus.

Quite what comment the Regent of the land of dreams might make upon the tactics of this game is beyond my reckoning. On the several occasions I have met that noble Lord it proved quite impossible to rouse him from his royal slumber for more than a few words of conversation in every hour. I am at a loss as to how one might keep him lucid for sufficient duration to play this excellent game.

I can only assume that the miserable cur has been adding Morpheus' tinctures to his wine and that they have addled his already meagre faculties.

So! Take another glass of port and tell me whether you fancy your chances!

(Listing 1 is checking whether the last card in a hand forms a winning trick.) ■

Listing 1

```
struct card
{
  enum suit_type{clubs, spades, hearts,
     diamonds};
  suit_type     suit;
  unsigned char value;
}

bool
match_suit(const card &a. const card &b,
   const card &c)
{
  return a.suit == b.suit && b.suit == c.suit;
}

bool
match_value(const card &a. const card &b,
   const card &c)
{
  return a.value == b.value &&
     b.value == c.value;
}

bool
mismatch_all(const card &a. const card &b,
   const card &c)
{
  return a.suit !=b.suit  && b.suit !=c.suit
     && c.suit !=a.suit && a.value!=b.value &&
     b.value!=c.value && c.value!=a.value;
}

bool
winning_hand(const std::vector<card> &hand,
   bool allow_mixed)
{
  if(hand.size()<3UL) return false;
  const size_t k = hand.size()-1UL;
  for(size_t j=1UL;i!=k;++j)
  {
    if(allow_mixed && mismatch_all(hand[0],
       hand[j], hand[k]))
    {
      return true;
    }

    for(size_t i=0UL;i!=j;++i)
    {
      if(match_suit (hand[i], hand[j],
         hand[k]) || match_value(hand[i],
         hand[j], hand[k]))
      {
        return true;
      }
    }
  }
  return false;
}
```

On a Game of Cards

A student performs an analysis.

Recall that the rules of the Baron's card game required that all but the number cards were discarded from the deck and that the remaining cards were given their face value with the proviso that black cards were to be considered positive and red cards negative. The play consisted of choosing 3 cards against which the Baron should then lay 3 of his own. The products of these 3 pairs of cards were then added together and, if the result had been zero, the Baron would have scored a point.

In the hand described, Sir R----- had received the cards 6♦, 2♣, 5♠, 4♥, 3♣.

When the Baron explained these rules to me I immediately recognised that it was essentially equivalent to reasoning about a vector space over an integer domain; specifically, about seeking a vector lying at a precise right angle to that represented by the cards lain down by the first player. Indeed, I said as much, but I fear that he may have misunderstood.

I must say that I found this a most ingenious puzzle since such problems are generally extremely resistant to the reckoning of men.

If we assume that Sir R----- played cards *a*, *b* and *c* from his hand and that the Baron responded with *x*, *y* and *z* then the result should have been

$$a \times x + b \times y + c \times z$$

Of course, the Baron's aim was that this sum should equate to zero, namely that

$$a \times x + b \times y + c \times z = 0$$

which, upon rearranging the terms, yields

$$x = -\frac{b \times y + c \times z}{a}$$

To minimise the chance that the Baron could have succeeded, I should have advised Sir R----- to choose such cards that *x* should have the least opportunity to take an integral value. To this end, I believe that it would have been most advisable to have chosen *a* such that it had the fewest common factors with the remaining pair, whatever they might have been; under this assumption 5♠ would clearly have been the best candidate.

Having so chosen, the Baron could thus have only scored a point if the sum of the remaining pair were a multiple of 5 and furthermore that the result of the above formula were less than or equal to 10. To minimise the range of numbers achievable with that sum it would have been sensible to choose the remaining 2 cards such that they have as many of their factors in common as possible; in this case either 2♣ and 4♥, 2♣ and 6♦ or 3♣ and 6♦.

Of these 3 choices, the last yields the greatest likelihood of large sums that one might expect result in integers greater than 10.

Whilst my reckoning has not been of sufficient rigor to qualify as proof, I should be most surprised if Sir R----- had contrived to play cards more likely to spoil the Baron's chances than 3♣, 5♠ and 6♦

AYE Conference Report

Jon Jagger gives a report of his experience at the AYE Conference.

The AYE (Amplify Your Effectiveness) conference was started by Jerry (Gerald) Weinberg about 10 years ago. Jerry was there but sadly he's very ill so only time will tell how many more he'll be able to attend. The conference is designed for people working in the software industry but aims to increase their effectiveness by increasing awareness at the personal and team levels.

This year's conference was held at the Embassy Suites hotel in Phoenix Arizona on Nov 9th, 10th, and 11th. The weather was hot and sunny as you'd expect in a desert city. The hotel is clean and spacious and air conditioned, the rooms likewise, and the staff friendly and helpful. It has a large heated outdoor pool with an accompanying jacuzzi, a spacious veranda area and an open-tent area for eating outdoors.

I've read that Jerry started the conference to win a bet he made whilst attending another conference he was not impressed with. It's therefore not too surprising that anything remotely resembling a PowerPoint presentation is banned and always has been. Instead the conference emphasizes simulation and experience.

JON JAGGER

Jon Jagger is a self-employed software coach-consultant-trainer-mentor-programmer who works on a no-win no-fee basis. He likes the technical aspects of software development but mostly enjoys working with people. He can be contacted at jon@jaggersoft.com

The website at http://www.ayeconference.com/ contains a wiki full of material spanning many years and is well worth a look if you are interested.

Each participant's name badge revealed their Myers Briggs personality type (you are asked to do an online test before you arrive). This provided an interesting topic of conversation but was only very lightly used during the scheduled sessions. Many of the sessions were role-play type games, typically organized into teams.

The conference is limited to 80 participants on a first come first served basis. $300 reserved a space and the total cost depended on how early you paid in full (reserve later and it's dearer). Paying at the earliest opportunity meant another $900 to pay. Plenty of drink and snacks are provided together with a buffet-style midday meal. On top of this the hotel room costs about $100 a night which includes an excellent breakfast. Add to this an evening meal and the flight.

The conference felt a lot like a non-technical version of the ACCU conference. It had a very relaxed atmosphere and yet at the same time was quite intense at times. I really enjoyed it and found it a very valuable experience. I met lots of great people and plan to attend next year. ■

Live to Love to Learn

Pete Goodliffe begins a journey of self-improvement.

Learning without thought is labour lost;
thought without learning is perilous.
~ Confucius

Programming is a creative, intuitive process. I, like many other programmers, like to think of it as an artistic pursuit; one of working a medium to produce something of utility and beauty. But all too often the commercial reality of coding is something more akin to being placed in a meat grinder, until every last ounce of coding prowess and motivation has been sucked from your soul.

Nonetheless, programming is an exciting and dynamic field to work in. One of the main reasons is that there is *always* something new to learn. Rarely are programmers forced to run around in circles performing the same repeated task for years and years, only discovering new ways to develop RSI and failing eyesight. We continually face the unknown: new problems, new situations, new teams, new technologies, or some combination of these. True, some programming jobs face more excitement than others, more of the unknown than others, and garner more techie thrills. But if you're going to be stuck in an office sat behind a desk, you may as well keep your mind occupied.

We are continually challenged to learn, to increase our skills and our capabilities. If you feel like you're stagnating in your career, one of the most practical steps you can take to get out of the rut is to take the effort to *learn something new*. On purpose.

Now, some people are naturally better at absorbing new information and 'getting up to speed' more rapidly than others. They are better at taking on new concepts and relating new concepts to gather a greater understanding. That's natural. But it's something we can all improve at. You need to take charge of your learning.

Check point

Ask yourself now whether *learning* is something that you think about consciously as a programmer. Is it something you consider as one of your programming skills? Do you actively try to learn things? Do you willingly put yourself into areas of the unknown? Or do you try to stick to what you know best, looking for an easy life?

Are you motivated as a programmer to find out new information and improve yourself? Do you relish learning? Or is it something of an inconvenience?

Do you want to improve as a programmer? I suspect that in this regard I'm preaching to the converted in this article. An ACCU member reading CVu clearly wants to widen their knowledge. Well done, give yourself a pat on the back! The simple fact is this: if you want to improve as a programmer, you *need* to be a skilled and seasoned learner. And you need to learn to enjoy it.

So let's think about the what, why and how...

Why learn?

Sometimes you have no choice but to learn; you are faced with a new task and you know nothing about the technology or the problem domain. You've got to get up to speed, and fast. Often the killer problem is that you have *no* time to do this in, and have to make a work estimate or deliver the first bit of functionality before you'd even have time to gather the vaguest overview of the topic.

The programmer's lot is not a happy one.

But even when you're coasting – working on reserve knowledge, with no need for new information – it's important to keep the grey cells ticking over and absorb new knowledge. One important reason is to simply cultivate a good learning habit; to maintain your ability to absorb information. Continually learning also helps to shape your programming attitude; to prevent you from believing that you're an expert! No one likes a know-it-all, after all.

There are plenty of other good reasons to develop yourself by learning. Your motivation might be to keep yourself fresh, to sharpen your existing skills, or simply to satisfy your natural curiosity. Or the reason might be more mercenary: to strengthen your employability, or allow you to manoeuvre into a programming field you're more interested in.

If you don't keep up a habit of continual learning you will go stale. You'll stagnate. The technological world will pass you by.

This demonstrates a simple learning maxim: **Learning. You've either *got* to, or you *ought* to**.

What to learn?

There's a whole world of things you could attempt to pick up. So what should you look at? Donald Rumsfeld summed up this conundrum in a particularly apt way when he made an infamous White House press conference: As we know, there are known knowns; there are things we know we know. We also know there are known unknowns; that is to say, we know there are some things we do not know. But there are also unknown unknowns – the ones we don't know we don't know.

So what do you pick? Clearly not something you know you know (although learning a topic more deeply, or attempting to consolidate your knowledge on a topic is a valuable task). Instead should you choose something that you *know* you don't know, or first learn about *what* you don't know in order to chose what to learn? That might make your brain bleed. Thanks a bunch, Rumsfeld.

Perhaps the list below will help. If you are learning for 'fun and personal profit' rather than your job leading you to a specific topic, you might consider:

- **Learn a new technology**. For programmers this is the obvious choice. We're fascinated by the different ways we can make electrons dance, and there's a wide field here to mine.

  You might choose to find out about a new programming language; there is no shortage of new and interesting languages being developed, there are many widely-used languages that you could learn to gain employability-enhancing skills, and there are many interesting existing languages. Consider looking at a language that promotes a different paradigm to your current languages to learn new ways to approach and solve problems – perhaps a functional programming language like Haskell would be a good choice. In their classic 1999 book, the Programmatic Programmers recommended learning one new language every year [1]. It's good advice. You don't have to become an expert, but do get beyond "Hello, World!"

  You could instead choose to learn a new library or application framework; perhaps an interesting low-level utility or a snazzy new

PETE GOODLIFFE

Pete Goodliffe is a programmer who never stays at the same place in the software food chain. He has a passion for curry and doesn't wear shoes. Pete can be contacted at pete@goodliffe.net

Another year, another great ACCU conference is coming!

Giovanni Asproni gives us a taste of things to come.

After the great success of ACCU 2009, which had roughly the same number of attendees of 2008 at a time were other conferences were closing for business, we had quite a challenge ahead. We wanted to organize a 2010 conference that was at least as good as, and hopefully better than, the 2009. Of course, we don't know if we have managed to achieve that just yet. However, I think that the programme we have created is a step in the right direction.

Selecting the programme for the conference is never an easy task, and when the number of proposal is very big – we had a total of 164 to select from, which is much more than the previous record set in 2009 – and the average quality is very good, the job of the committee becomes particularly difficult. Therefore, to make things a bit easier for us, and to make the most of the high quality proposals we received, we decided to add five more 90 minute slots, increasing the number from 55 to 60 and making the Saturday 90 minutes longer.

Some highlights include:

- A special track on software testing introduced by a pre-conference tutorial and a keynote both by James Bach
- A keynote and a Certified Scrum Master course by Jeff Sutherland, co-inventor of Scrum and one of the original signatories of the Agile Manifesto, held at the same time of the other pre-conference tutorials with a £100 discount on its standard price for ACCU members and conference attendees
- A keynote by Dan North, Agile development expert
- A keynote by Russel Winder, who doesn't really need an introduction to the ACCU crowd
- A pre-conference tutorial on the D programming language by the two creators, Walter Bright and Andrei Alexandrescu

And there is more! Just have a look at the conference web-site http://www.accu.org/conference to read the full programme and discover the roster of top-notch speakers.

I almost forgot. The booking price is the same as in 2009!

I'm looking forward to seeing you there.

Live to Love to Learn (continued)

high-level UI toolkit. After all, you can never have enough UI toolkits.

Or learn a new software tool. You should never underestimate the usefulness of learning new tools that will make your work more productive and/or more enjoyable. Learn how to use a new text editor, or IDE. Learn a new documentation tool or a test framework. Learn a new build system, an issue tracking system, a source control system (indulge yourself in the new *distributed version control* craze that all the cool kids are going on about), a new operating system, and so on.

Even if you are not going to use the new technology immediately in your 'day job', learning about it will almost certainly help you to use your existing technology in better ways, and will help you to evaluate new technologies when the need arises.

- **Learn new technical skills.** You might want to learn how to more effectively read code, or write technical documentation. You could learn how to manage a software team and climb the greasy career ladder.
- **Learn how to work with people.** Yes, this is tediously 'touchy-feely' for most code monkeys. But it can be an incredibly interesting field, and very useful. You could look at sociology, or study some management texts. This kind of information will help you to become a much more capable team worker, and will enable you to lead teams into directions that you think they should go in, rather than suffer in silence. It will help you to understand how people are communicating to you, and how to communicate effectively with them. You'll discover how to understand your customer better, and filter their requirements.
- **Learn a new problem domain.** Maybe you always wanted to write mathematical modelling software, or do audio DSP work. Without any experience or knowledge you'd be unlikely to fall into a job in a new sphere, so give yourself a head start and begin to learn about it. Then work out how to get practical, demonstrable experience.
- **Learn how to learn.** Seriously! Perhaps you could invest your time into finding out some new learning techniques that will help you absorb knowledge more effectively. Do you find there's a constant barrage of information you need to tap in to, and it just seems to flow past you? Investigate ways to seek out, consume, and absorb knowledge. There's a whole lot more to this than I can cover in these C Vu articles, after all! Consider learning and practising new skills such as mind mapping and speed reading.
- **Learn something completely different.** Or, more interestingly, you may prefer to choose something completely left-field with no relevance to your day job, and no obvious software applicability. Learn a new foreign language, a musical instrument, a new branch of science, art or philosophy. Hey, even spirituality. Far from being a pointless waste of time, or a distracting personal hobby, doing this will open your world view wider. If you're willing to look for the interesting themes and techniques you will certainly find that it helps to inform the way you program.

Above all, pick something that interests you. Pick something that will benefit you (the act of learning in itself is a benefit, but choosing something because it will give you fresh usable skills, broadens your insight, or brings you pleasure is a good thing). You will be investing a significant amount of time, so invest wisely! ■

Next time

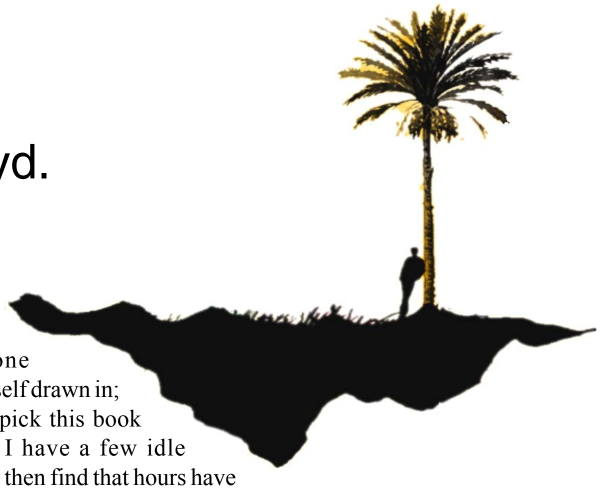
Having considered valuable things to invest your time in learning, we'll look at good learning techniques and how to sharpen our on-the-job information gathering skills. I hope that you find this diversion into a 'soft skill' programming topic, rather than my traditional technical ground, useful, interesting, and most importantly: motivating.

References

[1] *The Pragmatic Programmer*. Andrew Hunt and David Thomas. Published by Addison-Wesley, Oct 1999. ISBN: 020161622X

Desert Island Books

Paul Grenyer maroons Alison Lloyd.



I've been aware of Alison Lloyd on ACCU general for several years, but although I'm sure we both contributed to same of the same threads I don't think we really chatted directly until one year I decided I wanted an MP3 player with a digital radio and I was wondering if I could get a discount from the company she worked for. It turns out I couldn't, but I did learn what an excellent sense of humour Alison has!

Alison Lloyd

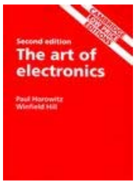
My first thought when asked to nominate some books I'd take to a desert island was to go for something like *Recognition and harvesting of edible plants on tropical islands*, or that well-known favourite, *Coconuts and sand: 50 recipes for shampoo*. However, once the format had been slightly more carefully explained, my collection came out as follows:

The C Programming Language

by Brian Kernighan and Dennis Ritchie, ISBN-10: 0131103628, ISBN-13: 978-0131103627

I'm a C programmer by trade, and this is the single reference book I refer to the most. Although C has some drawbacks, and is a bit dated by modern standards, I still find it elegant and a pleasure to use, particularly for the systems-level embedded work I do. It is unforgiving of mistakes and often produces somewhat cryptic code; nevertheless, a language whose entire structure and standard library can be described in 274 pages, that is almost universal in the platforms that support it, appeals to me. This is the seminal reference.

The Art Of Electronics



By Paul Horowitz and Winfield Hill, ISBN-10: 0521689171, ISBN-13: 978-0521689175

Originally written in the early 1980s, I first encountered Horowitz & Hill at university. This is one of the few textbooks that has stayed with me since then, through several moves and continents. As my interests have always been somewhere between software and hardware, this book provides an excellent reference, starting from the very basic and working up. It is written in a clear, understandable fashion, and although intended as a textbook (complete with exercises), it has information on pretty much everything to do with electronics. Of particular interest are the collections of 'Bad Circuits' – finding one's latest idea for an elegant solution to some minor problem already demolished is probably character building.

The Art Of Electronics was revised in 1989, so it is a little dated in places. That said, the fundamentals of electronics haven't really changed – recent advances have made many things easier and more convenient – so it's still a useful book to have handy.

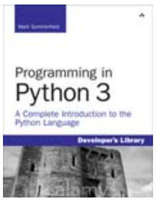
Principles Of Helicopter Flight

By Jean-Pierre Harrison, ISBN-10: 0-9638491-0-7

Sadly out of print, Harrison's book predates the much-better known work by W. J. Wagtendonk of the same name. It is aimed at the student helicopter pilot, but describes all the aerodynamics behind rotary-wing flight. It is written in an engaging style, and doesn't assume any prior aviation knowledge (although some basic maths and physics is useful). Helicopters are amazingly complicated machines, leading to some extremely interesting aerodynamics; Harrison manages to take something that should be dry and very mathematical and render it in a simple, easy-to-follow way, so that one finds oneself drawn in; I tend to pick this book up when I have a few idle moments, then find that hours have passed.

Principles Of Helicopter Flight is difficult to get hold of these days, although copies do surface in the usual secondhand book sources from time to time. There was a move afoot to reprint the book, but I'm not sure that ever came to anything.

Programming in Python 3



By Mark Summerfield, ISBN-10: 0137129297, ISBN-13: 978-0137129294

While not perhaps something I'd choose as a seminal work, I'm part-way through this book now. If nothing else, I'd hate to leave the book unfinished; whatever its faults, it is a pretty decent reference, and I'm enjoying Python. Unfortunately, it turns out that the main problem I'm likely to be solving with Python in the near future requires Python 2.x, so this may be a source of some frustration.

I really struggled to choose a single novel, so I stretched the definition a bit to include novel-sized items:

HTC Universal Smartphone

With solar charger, and 2GB SD card, containing most of Baen's back catalogue.

For those who've not run into them before, Baen are an American publisher, specialising in military scifi, but with good sidelines in hard science fiction and fantasy, as well as some esoteric stuff. Everything they publish in paper format is also made available in electronic form, with the latest books being sold via their website. US$15 gets you everything published in a given month (usually around 5 books); they also make many books and series available for free download.

The Universal has excellent functionality, and includes MS Reader as standard in most builds. It has a decent size screen with good brightness, and a conveniently-placed navigation rocker. I just love having a large library that's small enough (just!) to fit in my pocket – it means you always have something to do in those idle moments between other things. In addition, there are some quite good sudoku programs, and if all else fails, one can always use it to request a lift home (having had the foresight to be shipwrecked on a 3G-equipped island, of course).

Mr Pietersen & The Guys by The Mr Pietersen Band

The Mr Pietersen Band is an old-style Cape Town jazz band, consisting of banjo, cello base, sax and accordion. Having grown up in the Western Cape region of South Africa, this is music from my childhood and beyond, stretching back to the District 6 days. Completely instrumental, this music speaks to me of how people never lost hope, even through all the struggles of Apartheid and its aftermath. Life in South Africa is often quite difficult, even now, but people still manage to keep an amazingly good sense of humour.

ACCU London

James Lyndsay: Anticipating Surprises (or How to Find Problems and Persuade People You've Avoided Them).

James Lyndsay talked to ACCU London about testing. He started by bursting a balloon, and making those of a nervous disposition jump. This wasn't a surprise. We all expected it to happen, because he was armed with a large pin. Next James demonstrated three 'bugs' or surprises – one in a camera that left black bands around the edges of a picture when you zoomed in, another with a screen shot tool that made his screen go black when he tried to select a context sensitive menu, making it very difficult to bring in other apps into focus, for example his talk, and finally misinformation on the ACCU London webpage which had claimed the November talk would be happening in July. Fortunately the misinformation had been amended before the talk, so about 15 people turned up including some new faces.

James talked briefly about the difference between the requirements and behaviour of a system. The bugs are where these mismatch – the aim is to maximise the intersection of requirements and what's delivered. We then spent time testing a system – we were presented with an app which had four coloured buttons to press, and a few bug reports. For example 'If I press all four buttons it crashes'. So, some people tried pressing all four buttons – and it didn't crash. Other people got it to crash on their machines when they pressed four buttons. Others got it to crash by only pressing three buttons. Other people got it to crash by randomly pressing loads and loads of button in various orders and lost track of what they'd done. The exercise reminded us that sometimes state matters – what order did you do things in can make a difference. Bugs reports may tell you something it is necessary to cause the surprise, but that might not be sufficient. The audience postulated various explanations for what might be causing the crash – but most focussed on state. James 'fessed up in the end – the time between pressing just two different buttons caused the crash. As a tester you need to think laterally. Looking at the code can help you see potential problems. Unit tests will stop potential problems. However, there are many ways things can cause surprises – it takes a combination of tools, experience and creativity to find these. Testing is a creative and fun activity that deserves respect.

The talk presented many ideas about how to approach testing. As a final resort there's always Elisabeth Hendrickson's excellent Test Heuristic Cheat Mug available from http://www.cafepress.co.uk/testobsessed.101092273. You can find more details at http://testobsessed.com/

Further information on the interactive sessions is available at: http://www.workroom-productions.com/black_box_machines.html

Frances Buontempo

Desert Island Books (continued)

The Wall by Pink Floyd

I inherited my liking for Pink Floyd from my father, who actually managed to see them in concert, way back when. Multi-layered and scary, *The Wall* is perhaps the best-known Pink Floyd album, certainly with many of the 'standard' Floyd tracks. It may be clichéd, but it's still an album I keep coming back to; other Pink Floyd albums may be better polished and more mature, but the wall is still one of my all-time favourite albums.

Next issue: Terje Slettebø

What's it all about?

Desert Island Disks is one of Radio 4's most popular and enduring programmes. The format is simple: each week a guest is invited to choose the eight records they would take with them to a desert island (http://www.bbc.co.uk/radio4/factual/desertislanddiscs.shtml).

The format of 'Desert Island Books' is *slightly* different from the Radio 4 show. You choose about five books, one of which must be a novel, and up to two albums. Some people even throw in the odd film. Quite a few ACCUers have chosen their Desert Island Books to date and there are plenty more to go.

The rules aren't too strict but the programming books must have made a big impact on your programming life or be ones that you would take to a desert island. The inclusion of a novel and a couple of albums helps us to learn a little more about you. The ACCU has some amazing personalities and Desert Island Books has proved we only scratch the surface most of the time.

Each issue of CVu will have someone different. If you would like to share your Desert Island Books please email me: paul.grenyer@gmail.com.

ACCU Security 2009

ACCU Security 2009 was held on 7th November at Bletchley Park, home of the legendary World War II code breakers, and the site at which the world's first digital computer went operational. Subtitled *Yesterday, Today, and Tomorrow* it featured three invited speakers. Tony Sale, who established the first musuems in Bletchley Park in 1994, described how human operator errors enabled the Bletchley Park codebreakers to decipher vital German messages from both battlefield Enigma machines and the German High Command Lorenz cipher machines. Simon Singh talked about how he constructed the encrypted messages for the Cipher Challenge included in his history of cryptography *The Code Book*, and how the winners eventually cracked them. Phil Zimmerman, creator of PGP, discussed his new secure Voice-over-IP protocol ZRTP.

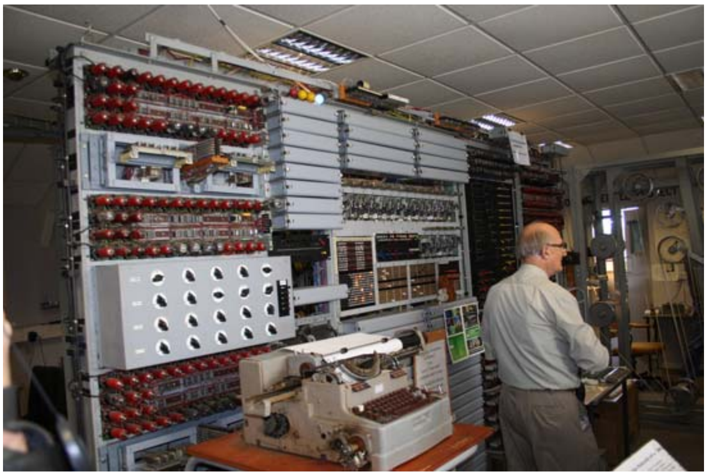
Bletchley Park is now a museum run by the Bletchley Park Trust. The Trust was formed in 1994, but wasn't able to secure control of the Park until 1999. By 2004, the Park was open to the public every day as a museum. The Trust was surviving, but only just. The site had been essentially unmaintained since the war and the buildings were in a desperate state of disrepair. In November 2008, a grant from English Heritage enabled vital repairs to be carried out. Milton Keynes Council went to a public vote over providing funding and the response was strongly in favour. The Trust has come a long way, but its finances still quiver on a knife-edge. ACCU held fund raising activities at the 2009 Spring Conference. Security 2009, organised by Astrid Byro Osborn and the conference committee, was held specifically to raise further funds. The event was a virtual sell-out, and raised over £5,250 for Bletchley Park Trust.

Simon Singh shows off his Engima machine.

Ralph McArdell

Part of the activities for the conference was a guided tour around the grounds of Bletchley Park including a look at the re-built Colossus – a hugely impressive feat and machine – and a tour of the computing museum. Both of these have moved since my previous visit a couple of years ago, when they were in temporary accommodation.

The computing museum exhibits in particular have changed for the better. On my first visit they were in a single hut all mixed up and not many were working. On the second only a few of the pieces were exhibited next to the tea room by the entrance. This time the museum had moved to more permanent accommodation and included a nice display of the smaller micro computing machines all arranged in cabinets or – for some – available to use. There were many comments anong our group along the lines of 'I remember these' as old friends were spotted amongst the displays.



Tony Sale with a replica of the Collosus Mk II that he built.

Our group then moved on to another couple of rooms where the larger old machines were displayed – the most obvious was an ICL beast previously used by Tarmac with quite a few of the large 'spin dryer' disk drives whirring away. Other exhibits of note were a couple of really huge displays showing air traffic control type data and a really old and interesting looking valve based beast being restored. Unfortunately I wandered by too late and missed most of what was said explaining what this machine was and how it worked – bother! I looked around for Prime mini computers as I had worked with these quite a lot in the 80s. I eventually found an anonymous Prime machine sitting next to a more interesting looking Cray of some sort in a room containing machines that had yet to be investigated.

I must go back for another look some time...

How many modern 2TB disks could you fit in the space one of these occupies?

Code Critique Competition 61

Set and collated by Roger Orr.

Please note that participation in this competition is open to all members, whether novice or expert. Readers are also encouraged to comment on published entries, and to supply their own possible code samples for the competition (in any common programming language) to scc@accu.org. A book prize is awarded to the winning entry.

Last issue's code

I'm trying to write a simple high precision integer class, using an array of chars as the representation. I've started, but have problems with my tests – the second output is wrong. I also want to be able to stream a bignum as a string, but I can't – any ideas?

Last issue's code is shown in Listing 1.

Critiques

John Bytheway <jbytheway@gmail.com>

Interface

Before delving into the implementation or tests for this code I think it's important to focus on the interface implied by `bignum.h`. Right now there are only three things supported: creating bignums from ints, converting them to strings, and adding them together. Minimalist, but enough to be interesting.

The most worrying piece of the header is the internal representation: just a **`std::vector<char>`**. Not part of the interface, but it has a worrying implication: With nothing but this to go on it's going to be very hard to represent negative numbers. After the variety of solutions to the negative number problem in CC 59 I don't want to second-guess the author of this code. I will assume that this class is only for non-negative numbers.

With this in mind, what constructors should **`bignum`** offer? Offering only the **`int`** constructor could lead to surprising behaviour when a larger integer is passed. It will be silently truncated. Better to take the largest possible type. In C++03 this means

```
  bignum(unsigned long = 0);
```

but that also could lead to surprising behaviour when a negative value is passed; we don't want them silently changed to positive numbers, so we should include

```
  bignum(long);
```

Unfortunately having both these constructors will cause ambiguity when initializing from smaller integer types. To avoid that we need to retain the existing constructor and add one more

```
  bignum(unsigned int);
```

ROGER ORR

Roger has been programming for over 20 years, most recently in C++ and Java for various investment banks in Canary Wharf and the City. He joined ACCU in 1999 and the BSI C++ panel in 2002. He may be contacted at rogero@howzatt.demon.co.uk

ACCU Security 2009 (continued)

Ewan Milne

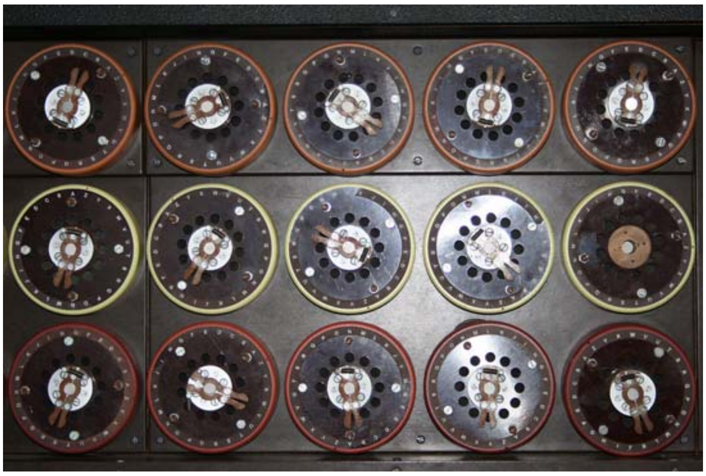
My (all too brief) career as an Enigma operator

The centrepiece of Simon Singh's talk was a demonstration of his own original Enigma machine. I was called up to help illustrate the symmetry of the Enigma cipher. Simon typed a two character message ('OK'), then reset the cogs to allow me to type in the coded message. The expected result, of course, was to see the lamps light up under 'O' and 'K' as the message was deciphered.

The actual result, of course, of my gingerly pushing the first key was... nothing at all. Well, this was a live demo, after all. After a second, more careful round of settings adjustment, the message was succesfully decoded with my role relegated to observer. And so I gained just a little sympathy for the hapless Enigma operators, whose errors had amused Tony so much during his talk.

Photographs courtesy of Anna-Jayne Metcalfe.

Old ICL mainframes.



Close up front panel of a Bombe.

Listing 1

```
-- bignum.h --
#include <string>
#include <vector>
class bignum
{
public:
  bignum( int i = 0 );
  operator std::string();
  bignum & operator +=( bignum const & rhs );
private:
  std::vector<char> value_;
};

-- bignum.cpp --
#include "bignum.h"

#include <algorithm>
#include <iostream>
#include <sstream>

bignum::bignum( int i )
{
  value_.push_back( i & 0xff );
  while ( i & 0xffffff00 != 0 )
  {
    i >>= 8;
    value_.push_back( i & 0xff );
  }
}

bignum::operator std::string()
{
  std::ostringstream oss;
  for (int idx = 0; idx < value_.size();
       idx++)
  {
    int v = value_[value_.size() - idx - 1];
    oss << std::hex << v;
  }
  return oss.str();
}
```

Listing 1 (cont'd)

```
bignum & bignum::operator +=(
   bignum const & rhs )
{
  value_.resize(std::max( value_.size(),
                rhs.value_.size()));
  char carry = 0;
  int idx = 0;
  for ( ; idx < std::min(value_.size(),
                rhs.value_.size()); idx++ )
  {
     value_[idx] += rhs.value_[idx] + carry;
     carry = value_[idx] < rhs.value_[idx];
  }
  for ( ; idx < value_.size(); ++idx)
  {
     value_[idx] += carry;
     carry = value_[idx] < carry;
  }
  if ( carry )
    value_.push_back(1);
  return *this;
}

-- test.cpp --
#include "bignum.h"
using namespace std;
int main()
{
  bignum b(0x1234567);
  b += 1;
  // won't compile: cout << b << endl;
  cout << b.operator std::string() << endl;
  b += 0x234;
  cout << b.operator std::string() << endl;
}
```

Note that all these constructors are introducing implicit conversions from builtin types to **bignum**. I think that this is appropriate, but there is still opportunity for surprises (such as implicit conversion from floating point types).

Next up is operator **std::string()**. Implicit conversion to string seems much more risky, and peeking ahead at `test.cpp` it looks like it only exists to make output streaming of **bignum**s work. But (as the author has observed) that won't work, and we'll need to introduce an output streaming operator

```
#include <iosfwd>
std::ostream& operator<<(std::ostream&,
   bignum const&);
```

I would probably replace the conversion to string with an explicit member function that converted to a specified base, but again I'll give the author the benefit of the doubt and assume that implicit conversion is needed (not to mention the fact that converting to most bases involves implementing division). But one thing that must change is that the conversion operator should be qualified **const**.

At this point the interface is still missing a lot, most obviously comparison operators, more arithmetic operators, and some way to convert a string to a **bignum**. More subtly, the implementation is such that this class is expensive to copy, so it should have an overload of swap and an assignment operator taking its argument by value to allow copy-elision on assignment from **rvalues**.

We'll ignore all of that too.

Test

Next I turn my attention to `test.cpp`. Things are not good here. To begin with, it doesn't compile without an additional **#include <iostream>** for **std::cout**.

But what about the cryptic comment about **cout << b << endl**? Indeed that doesn't compile (without my additions to `bignum.h` above). Why not? Because the overload of **operator<<**, though it is in scope (because namespace **std** has been pulled in) will not be found because it is a function template, and template argument deduction does not take into account implicit conversion operators. Nevertheless, it is not necessary to resort to explicitly calling the conversion operator; an explicit conversion or **static_cast** will do the trick:

```
cout << std::string(b) << endl;
```

But better still is to introduce an output streaming operator for **bignum** (as I mentioned above) and then the simple **cout << b** will work fine.

Diverting to `bignum.cpp` briefly to implement that streaming operator in in the simplest possible way

```
std::ostream& operator<<(std::ostream& o,
   bignum const& b)
{
  o << std::string(b);
  return o;
}
```

(ignoring a multitude of formatting questions: should this operator respect the stream properties with respect to base, padding, etc.?) it's now possible to compile a version of `test.cpp` unpolluted by calls to operator **std::string()**.

Running it reveals problems (surprise, surprise!), but it's not clear where in the implementation these problems are. A more detailed set of tests should tease these issues apart.

```
#include "bignum.h"
#include <iostream>
#include <cstdlib>
#include <limits>

using namespace std;

template<typename T>
void check_fails_to_construct(T i)
{
  try {
    bignum b(i);
    cout << "FAIL: bignum("<<i<<") constructed"
      "when it shouldn't" << endl;
    } catch (...) {}
  }
void check_conversion(bignum const& b,
  std::string const& v)
{
  if (std::string(b) != v)
    cout << "FAIL: " << b <<" != " << v << endl;
}
void check_sum(bignum const& b1,
  bignum const& b2, std::string const& v)
{
  // Check sum both ways round (addition is
  // commutative)
  bignum r1(b1);
  r1 += b2;
  if (std::string(r1) != v)
    cout << "FAIL: " << b1 << "+" << b2
         << " == " << r1 << " != " << v << endl;
  bignum r2(b2);
  r2 += b1;
  if (std::string(r2) != v)
    cout << "FAIL: " << b2 << "+" << b1
         << " == " << r2 << " != " << v << endl;
}
int main()
{
  check_fails_to_construct(-1);
  check_fails_to_construct<signed char>(-1);
  check_fails_to_construct<short>(-1);
  check_fails_to_construct<long>(-1);
  check_fails_to_construct(std::numeric_limits
    <int>::min());
  check_fails_to_construct(std::numeric_limits
    <long>::min());
  // Check can be constructed from all int types
  bignum(static_cast<bool>(0));
  bignum(static_cast<char>(0));
  bignum(static_cast<signed char>(0));
  bignum(static_cast<unsigned char>(0));
  bignum(static_cast<short>(0));
  bignum(static_cast<unsigned short>(0));
  bignum(static_cast<int>(0));
  bignum(static_cast<unsigned int>(0));
  bignum(static_cast<long>(0));
  bignum(static_cast<unsigned long>(0));
  check_conversion(0, "0");
  check_conversion(1, "1");
  check_conversion(0x7f, "7f");
  check_conversion(0x80, "80");
  check_conversion(0xff, "ff");
  check_conversion(0x100, "100");
  check_conversion(0x10000, "10000");
  check_conversion(0xffffffff, "ffffffff");
  check_sum(0, 0, "0");
  check_sum(0, 1, "1");
  check_sum(1, 1, "2");
  check_sum(0xff, 0x1, "100");
  check_sum(0xffff, 0x1, "10000");
  check_sum(0xffffff, 0x1, "1000000");
  check_sum(0xffffffff, 0x1, "100000000");
  check_sum(0xffffffff, 0x10000, "10000ffff");
  check_sum(0x80, 0x80, "100");
  check_sum(0x8000, 0x8000, "10000");
  check_sum(0xff80, 0x10080, "20000");
  check_sum(0xff, 0xff, "1fe");
  check_sum(0xffff, 0xffff, "1fffe");
  check_sum(0x1234567, 1, "1234568");
  check_sum(0x1234568, 0x234, "123479c");
}
```

Note that the implicit conversion to `std::string` doesn't work in `operator!=` either, even though all the template arguments could be deduced from the other argument. If anything, it looks like this conversion operator will confuse by not working, rather than working unexpectedly.

Running these tests promptly chews up all of my memory in the `bignum` constructor. Clearly it's time to look at the guts of this code...

Implementation

The first problem in `bignum.cpp` is `#include <iostream>`. This is defining far more than necessary, and may cause some performance penalty at start-up time. `#include <ostream>` is sufficient.

It's obvious from the constructor that the intended representation of the integer is one-byte-per-char, least-significant-first in `value_`. To get a unique representation requires some ruling about leading zero bytes (at the end of `value_`). The existing constructor will always put at least one byte in `value_`, and this might be convenient for the string conversion (because of the awkward case of 0) but it will probably cause grief in arithmetic operators, so I will impose the invariant `value_.empty() || value_.back() != 0`.

Now we have decided what the constructor should do, several issues are evident:

- When `i == 0`, it violates the invariant.
- It assumes `sizeof(int) == 4`. If it is larger then the condition of the `while` loop may fail to catch some large `int`s.
- It assumes right-shifting an `int` introduces zero bits, which may not be so (this caused the infinite loop in my tests).

Now is the time to switch from `int` to `unsigned long`, which solves the last problem. A slight refactoring of the loop solves the other two:

```
bignum::bignum(unsigned long i)
{
  for ( ; i; i>>= 8)
   value_.push_back(i & 0xff);
}
```

Someone following the letter of the standard might notice one remaining problem with this: the conversion from `unsigned long` to `char` is implementation-defined when `char` is signed and `(i & 0xff) > CHAR_MAX`. In practice it will probably do the 'right thing' on almost every platform, but this is just the first and most subtle of the problems of signed chars in this code. Changing `value_` to a `std::vector<unsigned char>` resolves this issue. Moreover, the desired behaviour is still guaranteed even if we delete the `& 0xff` above.

The other three constructors can be defined in terms of this one (with potentially some modest performance cost caused by swapping vectors):

```
#include <stdexcept>
bignum::bignum(long i)
{
  if (i < 0)
    throw std::invalid_argument(
      "negative bignums are not supported");
  bignum b(static_cast<unsigned long>(i));
  swap(value_, b.value_);
}
bignum::bignum(unsigned int i)
{
  bignum b(static_cast<unsigned long>(i));
  swap(value_, b.value_);
}
bignum::bignum(int i)
{
  bignum b(static_cast<long>(i));
  swap(value_, b.value_);
}
```

With those changes most of the output tests are working right, but some are still failing:

```
FAIL:  != 0
FAIL: 10 != 100
FAIL: 100 != 10000
```

It's no surprise to see that 0 isn't displaying properly since its internalrepresentation has changed, but there's clearly something else broken too.

The conversion operator is constructing the output one byte at a time by converting it to an `int` and using output streaming operators to a `std::ostringstream`. The change to `unsigned char` has already fixed the worst issue here: if `char` is signed then this code would output negative `int`s in hex mode, causing a string of extraneous `ff`'s. The remaining problems are:

- The special case of 0.
- When a byte is less than 0x10, it's printed as 1 hex digit instead of 2.
- `std::hex` is being sent with every byte when it is sufficient to send it once.

I also don't like the `int`-based indexing into the vector; better to use iterators. Here's an implementation that passes all the tests:

```
#include <iomanip>
bignum::operator std::string() const
{
  std::ostringstream oss;
  if (value_.empty()) {
    oss << '0';
  } else {
    std::vector<unsigned char>::
      const_reverse_iterator byte
      = value_.rbegin();
    // Most significant byte printed unpadded
    oss << std::hex << std::setfill('0')
        << int(*byte);
    ++byte;
    // Remaining bytes padded to a width of 2
    for ( ; byte != value_.rend(); ++byte) {
      oss << std::setw(2) << int(*byte);
    }
  }
  return oss.str();
}
```

Note that it is important to convert the unsigned `char`s to `int`s before streaming them, because otherwise they'll be rendered as `char`s rather than hex integers.

Finally, we have the addition operator. Only two of the tests are failing (and I confess I only added these tests after I saw the bug in the code):

```
FAIL: ff80+10080 == 10000 != 20000
FAIL: ffff+ffff == fffe != 1fffe
```

This is due to an error in the carry detection in the first `for` loop. It says

```
value_[idx] += rhs.value_[idx] + carry;
carry = value_[idx] < rhs.value_[idx];
```

when it should be

```
value_[idx] += rhs.value_[idx] + carry;
carry = value_[idx] < rhs.value_[idx] + carry;
```

The first version fails when `carry == 1` and `value_[idx] == 255`. In this case `value_[idx]` is changed by the first line to equal `rhs.value_[idx]`, so the second line doesn't detect a carry when clearly one has happened. Note that the new version only works because the sum `rhs.value_[idx] + carry` is computed as an `int` (rather than some kind of `char`), and thus can take the value 256.

With that change the tests all pass, but there's still one bug and a couple of optimizations that look sensible:

- `idx` is only an `int`. If `value_.size() > INT_MAX` then it will overflow (though this event is essentially impossible to achieve with the existing interface).
- `idx` should be a `size_t`, or the whole thing should be rewritten using iterators.
- The condition in the first `for` loop can be simplified. Thanks to the resize we know that:
  `value_.size() >= rhs.value_.size()`
  so there's no need to call:
  `std::min(value_.size(), rhs.value_.size()):`
  the result will be `rhs.value_.size()`.
- The second `for` loop can be cut short early if ever `carry == 0`. This will for example make incrementing large numbers much faster.

C++0x

This class is a good example of how things should change in the light of C++0x. For example, we can:

- Change unsigned `char` to `std::uint8_t`, which clarifies the intent of the code.
- Add a `move` constructor to avoid more expensive copies (a move assignment operator is not needed if there is already an assignment operator taking its argument by value).
- Add constructors taking `long long` and `unsigned long long` arguments.
- Make the `string` conversion operator explicit, thus avoiding user confusion when it fails in circumstances like the use in `operator<<` or `operator==` (but gcc doesn't support this C++0x feature as of 4.4.1).

Furthermore, if six constructors are deemed too much code duplication, they can be replaced by two (and the standard default constructor) using the new type traits, at the price of making them less readable and moving the implementations into the header.

With all of that, the header becomes:

```
#include <string>
#include <vector>
#include <iosfwd>
#include <cstdint>
#include <type_traits>
#include <stdexcept>
class bignum
{
public:
  bignum() = default;
  // Constructor for unsigned types
  template<typename T>
  bignum(T i,
      typename std::enable_if<
        std::is_unsigned<T>::type::value,
        int>::type = 0) {
    for ( ; i; i>>= 8)
      value_.push_back(i & 0xff);
  }
  // Constructor for signed types
  template<typename T>
  bignum(T i,
        typename std::enable_if<
          std::is_signed<T>::type::value, int
        >::type = 0) {
      if (i < 0)
        throw std::invalid_argument(
          "negative bignums are not supported");
      bignum b(static_cast<typename
        std::make_unsigned<T>::type>(i));
      swap(value_, b.value_);
    }
    // Move constructor
    bignum(bignum&& b)
    : value_(std::move(b.value_)) {}
    // Assignment operator (good for copies or
    // moves)
    bignum& operator=(bignum b) {
      swap(value_, b.value_);
      return *this;
    }
    operator std::string() const;
    bignum& operator+=(bignum const& rhs);
  private:
    std::vector<std::uint8_t> value_;
  };
  std::ostream& operator<<(std::ostream&,
     bignum const&);
```

All of the C++0x features used here are supported by gcc 4.4, and all the tests still pass (although more should probably be added to exercise the new features).

Conclusion

There was quite a lot to say about this code, and the hard work (implementing multiplication, etc.) has yet to begin. I would advise the author of this code in the strongest terms to abandon the effort to write a **bignum** class, and use a pre-existing one instead. It should provide all the necessary features and be much more reliable and faster than anything homegrown.

Commentary

I think the most important remark in the critique above is the conclusion: use a pre-existing class (if possible). The code demonstrates a very simple interface to a big number class and I'm sure a user of the class would soon discover the lack of many useful methods.

Using a pre-existing class leaves you more time to get on with the problem the bignum class was designed to help with.

The Winner of CC 60

John provided a pretty exhaustive critique of the code and well deserves the prize!

Code Critique 61

(Submissions to scc@accu.org by Feb 1st)

I am trying to write a simple two-way map to allow items to be addressed by two different keys but I'm getting some odd behaviour. I've written a simple test program and everything works fine except for the last line output – I expect the last size to be 1 but I get 2. Can you help?

The code is in Listing 2. You can also get the current problem from the accu-general mail list (next entry is posted around the last issue's deadline) or from the ACCU website (http://www.accu.org/journals/). This particularly helps overseas members who typically get the magazine much later than members in the UK and Europe.

Reflection on Code Critique 60

Martin Moene <m.j.moene@eld.physics.LeidenUniv.nl>

Proto-pattern: Machine-tested Program Text[1]

Context As an author you are drawing attention to certain properties of software. You illustrate this with example code.

Problem Books and articles are proofread and corrected to prevent errors in them as far as possible, however this does not guarantee error-free texts. Errors in code examples must not occur as they may confuse a reader who is not yet sufficiently familiar with the subject at hand. Unlike non-code text, the software examples can be executed and tested for correct behaviour mechanically.

Forces t.b.d.

Solution Test the example code with a unit test and present the tested code in the article or book.

Positive Consequence Provided with error-free examples, your readers may have an easier time to understand the point you are making.

Negative Consequence Some people may become less alert reading a text when they cannot spot an occasional error.

Known Uses In *The Practice of Programming*, Kernighan and Pike write: '... We've tried hard to write our own code well and have tested it on a dozen of systems directly from the machine-readable text.' [2a]

In *The C Programming Language*, Kernighan and Ritchie write: 'As before, all examples have en tested directly from the text, which is in machine readable form.'[2b]

In *The AWK Programming Language*, Aho, Kernighan, and Weinberger write: 'The examples have all been tested directly from the text, which is in machine readable form.' [2c]

Well, a clear pattern emerges here.

Related to this are texts created with a literal programming system (http://en.wikipedia.org/wiki/Literate_programming), such as [2d] and [2e].

Discussion I'm not sure if we may call this a proto-pattern or that it's merely a good practice forced into pattern form[3]. Although presented as a stand-alone pattern, it could also be part of a pattern language for authoring software articles. Maybe one even already exists, even though I did not find one.

Notes and references

[1] Inspired by **#include <iostream>** missing from `test.cpp`; as the author mentions shortcomings of the program's output ('the second output is wrong'), it must have been created from a different version than from the source code shown (or it must be that **<string>** or **<vector>** of the compiler used by the author has the side-effect of making **std::cout** available; MS VC8 does not do that).

[2a] Brian W. Kernighan and Rob Pike. (1999). *The Practice of Programming*. Addison-Wesley. p. xi.

[2b] Brian W. Kernighan and Dennis M. Ritchie. (1988). *The C Programming Language*. Prentice Hall, second edition. p. ix.

[2c] Alfred V. Aho, Brian W. Kernighan, and Peter J. Weinberger. (1988). *The AWK Programming Language*. Addison-Wesley. p. v.

[2d] Christopher W. Fraser and David R. Hanson. (1995). *A Retargetable C Compiler, Design and Implementation*. The Benjamin/Cummings Publishing Company, Inc.

[2e] Allen I. Holub. (1990). *Compiler Design in C*. Prentice Hall, second edition.

[3] Frank Buschmann, Kevlin Henney, and Douglas C. Schmidt. (2007). *Pattern Oriented Software Architecture: On Patterns and Pattern Languages*, Volume 5. John Wiley & Sons.

Listing 2

```
twowaymap.h
#include <map>
template <typename T, typename U>
class twoway_map
{
public:
  typedef typename
  std::map<T,U>::size_type size_type;
  void insert(const T& key1, const T& key2,
    const U& value)
  {
    U & p = map1[key1] = value;
    map2[key2] = &p;
  }
  void erase1(const T& key1)
  {
    if (map1.find(key1) != map1.end())
    {
      for (typename std::map<T,U*>::iterator
        it = map2.begin();
        it != map2.end(); ++it)
      {
        if (it->second == &map1[key1])
        {
          map2.erase(it->first);
          map1.erase(key1);
        }
      }
    }
  }
  void erase2(const T& key2)
  {
    if (map2.find(key2) != map2.end())
    {
      for (typename std::map<T,U>::iterator
        it = map1.begin();
        it != map1.end(); ++it)
      {
        if (&it->second == map2[key2])
        {
          map1.erase(it->first);
          map2.erase(key2);
        }
      }
    }
  }
  U& at1(const T& key1)
  {
    return map1[key1];
  }
```

Listing 2 (cont'd)

```
  U& at2(const T& key2)
  {
    return *map2[key2];
  }
  size_type size() const
  {
    return map1.size();
  }
private:
  std::map<T,U> map1;
  std::map<T,U*> map2;
};
test.cpp
#include "twowaymap.h"
#include <string>
#include <iostream>
struct colour
{
  colour() {}
  colour(int r, int g, int b)
  {
    rgb = r << 16 | g << 8 | b;
  }
  operator int() { return rgb; }
  int rgb;
};
using std::cout;
using std::endl;
int main()
{
  twoway_map<std::string, colour> m;
  m.insert("Red", "Rouge", colour(255,0,0));
  m.insert("Green", "Vert", colour(0,255,0));
  m.insert("Blue", "Bleu", colour(0,0,255));
  cout << "size: " << m.size() << endl;
  cout << std::hex << m.at1("Red") << endl;
  cout << std::hex << m.at2("Vert") << endl;
  cout << std::hex << m.at1("Blue") << endl;
  m.erase2("Bleu");
  cout << "size: " << m.size() << endl;
  m.erase1("Blue");
  cout << "size: " << m.size() << endl;
  m.erase1("Red");
  cout << "size: " << m.size() << endl;
}
```

Write for us!

C Vu and Overload rely on article contributions from members. That's you! Without articles there are no magazines. We need articles at all levels of software development experience; you don't have to write about rocket science or brain surgery.

What do you have to contribute?

- What are you doing right now?
- What technology are you using?
- What did you just explain to someone?
- What techniques and idioms are you using?

For further information, contact the editors: cvu@accu.org or overload@accu.org

Bookcase

The latest roundup of book reviews.

If you want to review a book, your first port of call should be the members section of the ACCU website, which contains a list of all of the books currently available. If there is something that you want to review, but can't find on there, just ask. It is possible that we can get hold of it.

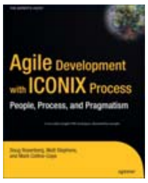
After you've made your choice, email me and if the book checks out on my database, you can have it. I will instruct you from there. Remember though, if the book review is such a stinker as to be awarded the most un-glamourous 'not recommended' rating, you are entitled to another book completely free.

I must thank Blackwells and Computer Bookshop for their continued support in providing us with books.

Jez Higgins (jez@jezuk.co.uk)

Agile

Agile Development with ICONIX process



By Doug Rosenberg, Matt Stephens and Mark Collins-Cope, published by Apress (2005), 261 pages, ISBN: 978-59059-464-3

Reviewed by Derek Graham

Many people with be familiar with two of the authors of this book from their earlier critique of *XP – Extreme Programming Refactored*. I was looking forward to this book because of the subject matter and the book's subtitle: People, Process and Pragmatism. The back cover also mentions that it follows a real life project from requirements to actual code and a working application.

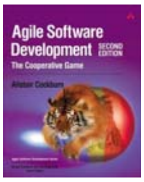
The book's stated aim is to cut through what it portrays as the hype of agile development and offer advice on finding a compromise between agile techniques and a what it calls 'disciplined' approaches. The book is well laid out, has clear diagrams and has lots of sections where each of the authors get to 'discuss' their individual take on a practice or technique and offer their own advice and insight.

We start with a couple of chapters which introduce agile and the author's ideas of what makes for a good software process. Once this is out of the way, they quickly move onto something called the ICONIX software process and this is where the book started to lose some credibility for me. In the days when Rational introduced their own RUP (Rational Unified Process), the ICONIX process was featured in articles by Doug Rosenberg as being the best bits of RUP and UML and some extra modelling techniques. Now it seems that because agile is flavour of the month, ICONIX is actually the best bits of agile, with UML and some other modelling techniques. Quite a bit of the book is therefore devoted to extolling the virtues of this process and why a subset of UML and agile techniques is what you should be adopting.

Later chapters introduce the mapping application and discuss the high level requirements and how they should be modelled. The authors do a good job of describing the prototype design with UML diagrams and snippets of C#, showing how the design changes through various versions and how the changes are incorporated release planning and iterative development. Most of this is done from the use case level and working downwards into sequence diagrams and class diagrams before showing small pieces of source code. Despite being a book about agile, TDD is left out of the process and put into a chapter at the end of the book where it is covered in a very superficial way as a 'Vanilla' TDD process. This is immediately followed up with a chapter on doing TDD the ICONIX way.

Overall, I think this has been a very creditable, brave effort to portray some real life aspects of an agile project but can't help feeling it loses something in being a 'special' sort of agile. In my opinion it would have been even better sticking with a more widely accepted process and devoting more of the book to illustrating the techniques in greater depth. All together I found it not very inspiring and think that the intermediate developers that this book is aimed at may find better agile texts elsewhere.

Agile Software Development, 2nd edition



By Alistair Cockburn, published by Addison Wesley (2006), 504 pages, ISBN: 978-321-48275-4

Reviewed by Gail Ollis

If you have read my Desert Island Books you won't be surprised to learn that this book, one of my selections, comes highly recommended. It speaks to me because it emphasises the very human characteristics that make psychology every bit as crucial to software development as technology. Actually, I'd contend that the issues affecting projects are at least 95% psychological, but if you haven't read the book yet you might feel that this overstates it. Let's just agree that human behaviour is an important factor; reading Cockburn helps to underline why I feel so strongly about it.

The need for communication is a very human issue that affects any software beyond that which you write on your own for fun; there's no problem to solve when it's just you and a beer (or mug of coffee, or other programming beverage of your choice). Code is pretty unambiguous; the computer will do what you told it, however much it may sometimes seem otherwise. Expressing your ideas using natural language, to another unpredictable human rather than a machine, is a different and difficult matter which Cockburn devotes his first four chapters to exploring. After describing this problem of 'managing the incompleteness of communications', he develops it further by comparing the nature of software development to a team co-operative game requiring invention and communication. The characteristics of individuals and those of teams are both significant factors in such a game, so each of these has its own chapter. The issues addressed in this first half of the book apply beyond the domain of software development, but being generalisable doesn't mean there is a lack of specific and practical advice to tackle them. My personal favourite is the 'information radiator' (a display of information positioned where passers-by will see it); an idea as simple as it is effective, of value regardless of whether your project is Agile and even regardless of whether it's Software Development.

I strongly believe that it is essential to know what problem you are trying to solve before

Bookshops

The following bookshops actively support ACCU (offering a post free service to UK members – if you ever have a problem with this, please let me know – I can only act on problems that you tell me about). We hope that you will give preference to them. If a bookshop in your area is willing to display ACCU publicity material or otherwise support ACCU, please let us know so they can be added to the list

- **Holborn Books Ltd** (020 7831 0022)
  www.holbornbooks.co.uk
- **Blackwell's Bookshop**, Oxford (01865 792792)
  blackwells.extra@blackwell.co.uk

applying any measures to address it. The first four chapters are an excellent way of answering this question for anyone thinking of applying a methodology measure, and so provide the ideal context for the following chapter on analysing, designing and refining methodologies. However, the book is designed to cater for a wide range of readers. I don't think any reader engaged in the world of software development (or indeed other collaborative enterprises) could fail to benefit from reading these quite general early chapters, whether or not the later chapters interest them, but readers who have already understood these problems can easily pick up the book from chapter 4 to review the specific principles they need to keep in mind when choosing a methodology. Equally, they can go straight to chapters 5 or 6 if they are looking for information about agile methodologies in general or the Crystal family of methodologies in particular.

This book is now in its second edition. Like me, many of you will have read it in its first edition; Cockburn has spared us the frustration of trying to find out what's new by leaving all the original chapters intact except for corrections. The evolution of his ideas since the first edition is instead laid out in a chapter n.1 for every chapter, and these are kindly marked with grey tabs on the page edge so that flicking through the book to find them is made very easy. Like so many of the ideas in this book, this is a simple yet effective solution to a common frustration.

The 'evolution' chapters do not contradict the content of the originals – if they did, there wouldn't be an advantage to this approach! Rather, they bring things up to date with the current state of the craft in light of the events over the intervening years. Thus by far the largest update is for the chapter on agile principles. It answers many persistent questions about practical aspects of the agile model and how it fits with others. It also has to dedicate nearly 20 pages to debunking pervasive myths that have developed about agile over the years – a sad sign, perhaps, that 'agile' has achieved greater penetration as marketable rhetoric than as a genuine implementations of the core principles that make it work.

This excellent book provides guidance rather than dogma, explaining what you might choose to do and why you would choose to do it and positively encouraging teams to ask themselves these questions. It describes agile as 'an attitude, not a formula' and not only fosters that attitude, but provides some pragmatic tools for putting it into practice.

Agile Project Management: Creating Innovative Products, 2nd Edition

By Jim Highsmith, published by Addison Wesley (2009), 432 p ages, ISBN: 978-0321658395

Reviewed by Allan Kelly

The fact that a book reaches a second edition is always a good sign. Jim Highsmith's *Agile Project Management* book was first published in 2004, and according to the bibliographic information in the second edition this one is published in 2010. Given that it reached the ACCU in mid-2009 there is obviously some schedule padding going on.

I must confess I never read the first edition so I can't comment on whether this edition is vastly improved or even changed. What I can say is that it is worth reading if you are involved with the management of Agile projects or just want to get beyond the iterations and TDD side of things.

Whether because management was ignored (XP), was seen as unnecessary (Scrum) or simply because Agile as we know it is heavily developer centric, the role of management in Agile development has not received enough attention. This book sets about putting management back on the Agile table. Highsmith is equally dismissive both of those who believe self-organization means no management, and those who are overly hierarchal and wedded to command and control style of management.

Within Agile, and Scrum specifically, a lot of the talk about self-organizing teams has left the management out in the cold. Highsmith addresses this directly. He rebalances the argument explaining why organization – and therefore management – is needed to so that self-organizing teams can be effective. It doesn't happen by accident.

His solution is (drum roll): Leadership. Agile teams, he says, need to be lead to delivery.

Whether you are a manager, a leader, developer or someone else who needs to get an Agile team to delivery you will find good information here. You will also find chapters on Governance, Scaling up Agile, distributed teams and define price/time/requirements projects.

Overall the book it is a bit long and Highsmith does wander a bit. Rather than sticking to his main point he does seem to take diversions. Although he usually returns to his point the diversions add length. Some of the typesetting could be better, his side boxes, diagrams and exhibits often use a smaller font than the rest of the text which makes them more difficult to read and funny looking.

Another disappointment is that he doesn't make more of the difference between in-house (corporate IT) development and software products. He is more aware of this than most authors but moves between the two a little too easily. For an otherwise comprehensive book this could have been looked at in more depth. But then, the book is sub-titled 'Creating Innovative Products' so perhaps this wasn't his main concern.

These are minor reservations and I highly recommend this book to anyone who is trying to understand where managers add value in Agile work.

iPhone

iPhone Cool Projects

By Gary Bennet et al, published by Apress (2009), 209 pages, ISBN: 978-1-4302-2357-3

Reviewed by Pete Goodliffe

Recommended

iPhone programming is one of the current 'hot topics' and we're seeing an increasing number of books published on this topic. This one is a bit of a mixed bag.

This is not an introductory tome; it requires significant prior understanding of the iPhone toolset and development environment. Instead, the book presents a number of complete fully-working iPhone applications covering various core iPhone technologies. It fits into a series of other Apress iPhone titles. Not having seem the other books, I can't say how well it complements the other titles in the series.

The book is effectively a collection essays by many authors, one per chapter, all 'experts' at various aspects of iPhone development. Some of them have produced very successful iPhone applications.

The topics covered are: simple game programming, peer-to-peer networking, multi threaded applications, creating multi-touch interfaces, physics and 2D animation libraries, audio streaming, and creating a location-aware application with navigation-based UI. There are no topics covered that you can't fathom relatively easily from the free Apple documentation and a bit of careful thought. However the useful piece of the jigsaw is seeing how other developers have already learnt iPhone OS and solved the common problems.

The production quality of the book is high. It has been very well presented, in full colour throughout, with many iPhone and Xcode screenshots. On the whole, the writing is good. Some chapters appear to have been better proof read than others.

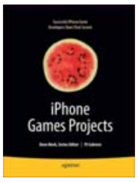
Perhaps the most useful part of the book is the availability of the source code for all the sample applications (from the publisher's website), so you can run and take apart the projects at your leisure. There is no bundled CD, and I'm more than happy with that.

As with many such multi-authored books, some chapters are better than others. Each chapter is relatively short, and they all basically provide an overview of their topic – enough to pique your interest, but not enough to answer any serious questions. For some topics this works better than others.

Highlights are the first game-writing chapter, the multi-touch interface chapters, and the location-based application chapter. These present useful information about how to write a 'real' iPhone application. I felt let down by the threading chapter which presents a fairly glib and un-thorough overview of the perils of writing threaded apps. The networking chapter is a very simplistic introduction; nothing it says is wrong, but to write your own serious networked application you'd really need to know a lot more about network technologies.

If you're an experienced programmer who wants a casual introduction to some more meaty iPhone projects than you've seen in the introductory tests, this book may be interesting for you. It's easy to read, fast paced, and pretty.

iPhone Games Projects



By PJ Cabera, published by Apress (2009), 258 pages, ISBN: 978-1430219682

Reviewed by Pete Goodliffe

Verdict: OK

This is the second book I've reviewed in this Apress iPhone series (the first being *iPhone Cool Projects*). The book has many of the characteristics of the first: it is full-colour throughout, contains clear writing, beautiful presentation, and relatively good copy editing. It hangs together about as well as the other book, too, which is 'mostly'.

It is a series of 8 distinct essays by different 'experts' (a relative term on such a new platform) iPhone game developers. The tone and approach of each chapter is therefore different.

The collection of topics covered is OK, but doesn't spread over the entire broad spectrum of game topics: there are TWO essays on networking, TWO essays on optimisation, one on multi-platform development (interesting in an 'iPhone' book), one on writing a design document, and a walkthough of a simple board game.

There are many more topics that might have been interesting chapters to have in this type of book: a 3D graphics primer, when/how to select a third party games engine, considerations for getting your game noticed in the app store, and more.

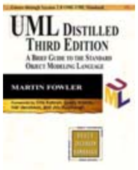
There are some recurring themes: a few authors suggest prefering C over Objective-C (for obvious reasons). There is some discussion of why C is 'better' than C++ which is (to a C++ programmer) unbalanced, and misleading.

As ever, the source code to each project is available from the Apress website. The quality of some of the code is quite variable.

If you want to write an iPhone game this book might be an interesting read, but I wouldn't suggest that every iPhone game programmer HAS to buy it. Some sections of it have far more value than others. In fact, I think overall you'd get more milage from the *iPhone Cool Projects* book since it covers a broader range of topics. I'm left feeling that the two books rolled into one would probably have been a better product. And I'm still not convinced that the title is even gramatically correct.

Miscellaneous

UML Distilled, 3rd Edition



By Martin Fowler, published by Addison Wesley (2003), 208 pages, ISBN: 978-0321193681

Reviewed by Chris Oldwood

I originally bought a book on UML back in 1998 after someone downloaded a demo version of the Select CASE tool. I thought UML was really going to take off as it seemed to provide a comprehensive language for describing software systems. Fast forward 10 years and my experience hasn't changed radically since those first doodles – most other developers still don't seem to use any sort of modelling, either as a documentation or design tool. Consequently I have always struggled when putting together various diagrams as I've never know whether I'm doing it right: too much detail or too little, aggregation or association, stereotype or parameterised type ...

UML Distilled, for me, answers many of those questions. As Fowler describes right from the start, some people do use UML as a blueprint to guide construction, but he, along with many others only use it mostly for sketching. This sets the tone for the rest book as he focuses on those aspects of the language that provide the most bang-for-buck when using UML in this way. However at each juncture, should you want to know more, he points you in the direction of the necessary resources; in this way the book would also act as a good introduction to the material.

Whereas my previous book walked through the UML in a waterfall-esque fashion covering requirements then static structure through to deployment; Fowler guides you based on his use cases [pun intended]. This means he covers Class Diagrams first; in fact he discusses object, package and deployment diagrams before getting to Use Cases. Admittedly Deployment Diagrams do only get 2 pages, but they aren't exactly the most complicated topic to digest. When he does finally tackle Use Cases I found it reassuring to find him pointing out that the real value is in the Use Case text – the diagrams often provide little additional benefit.

State Machine and Activity Diagrams then get 10 or so pages each, with the latter getting extensive coverage because of its new lease of life in UML 2.0, and it has certainly piqued my interest due to its ability to illustrate parallel workflows. He then goes on to briefly cover the various other diagram types, such as the renamed Communication Diagrams (previously known as Collaboration Diagrams), Component Diagrams, Timing Diagrams etc. I was a little disappointed he didn't spend more time on Communication and Component Diagrams (he suggests that people generally prefer Sequence Diagrams to the former) as they both looked promising.

Although the book tips the scales at less than 200 pages, he still manages to cover the history of UML, a section on development methodologies, and an appendix to summarise the key changes from 1.0 through to 2.0.

An Introduction to Programming with C++, 4th Edition

By Diane Zak, published by Course Technology (2005), 605 pages, ISBN: 9780619217112

Reviewed by Giuseppe Vacanti

I start writing this review a few months before the scheduled release of the sixth edition of this book, this review covering the fourth edition from 2005.

This is not a book I have liked a lot. First of all the title troubles me, but this is probably due to my not being a native: I thought the book would be an introduction to C++, whereas it turns out to be an introduction to programming, where C++ plays the role any other programming language could have played.

There is no mention of the Standard Template Library, for instance, and therefore there is no mention of vector, and the only arrays we see are the plain C-style arrays. Given the almost 600 large format pages, one could have expected a bit more. We do encounter the transform algorithm and strings, however, although the transform algorithm is introduced rather empirically to convert a string to upper case without mentioning the word iterator.

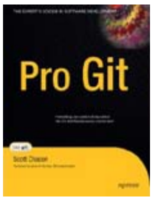
The book begins by explaining what a computer is, and how its main components are connected to one another. It goes on to explain the problem-solving process, how to break a problem down to arrive to an algorithm, and finally how to start coding this algorithm in C++. The pace is slow, each section or chapter ends with many questions and problems with answers.

We move through control structures, flow charts, input and output, and functions, and close to the end of the book we reach strings, where things start getting C++-ish. The book ends with a chapter on classes, but we run out of pages before we can understand how classes can be used to achieve a greater level of abstraction.

The book is not challenging in any way, and this is perhaps the author's intent: everything is spelled out and numerous exercises bring all the points home. From this point of view the book could be interesting for somebody who has never come close to programming before. The book is complemented by a web-based 'Testing Center' that the reader can get access to through the access code provided with the book. My version also came with Microsoft Visual C++ .net 2003 student edition, and it can be used to compile the test programs that can accompany the book and can be downloaded from the Testing Center.

In summary, this is a book for extreme beginners. It has no obvious flaws, but I found it not challenging in any way, and for this very reason I would not recommend it.

Pro Git



By Scott Chacon, published by Apress, 265 pages, ISBN: 978-1430218333

Reviewed by Pete Goodliffe

Highly Recommended

It's not often I start a book review with glowing praise. This time, I will: if you use the git version control system, or are thinking of using git in the future, get this book. It's excellent.

Pro Git available online from http://www.progit.org (or git clone the book's source from http://github.com/progit). This means that you can read it for free before considering a purchase. Indeed, that's where I started. However, I *highly* recommend the dead tree version. Apress' production quality is excellent and the paper copy is definitely a valuable thing to have.

The book is an excellent introduction to using git; it's perfect for newbies, and a good reference for existing users. It starts from first priniciples. That is, it describes what git is, and what a distributed version control system is. It briefly introduces version control in general, but that is really prerequiste information.

The text is well paced, and very clearly written. The examples are well chosen and the coverage of git's facilities is broad.

The author starts with installing/configuring git and outlines the basic git principles. He covers basic operations (check in, clone, viewing logs, tagging). Then he moves onto git's crowning glory: branching and merging. This potentially tricky topic is covered very well.

The book also covers running a git server, sensible workflows to tame distributed collaboration, useful/advanced git facilities (stashing, amending history, binary searches, subtree merging, client- and server-side hooks), and using git with other version control systems. In particular, there is good coverage of using git as a more advanced subversion client.

The final chapter is particularly useful: a great overview of git internals. This sounds relatively pointless when you've covered most git usage already. However, this is a great chapter – the author explains what's going on under the covers in such a way that you gain a much better insight into how all the high-level git operations work.

The Old New Thing

By Raymond Chen, published by Addison Wesley (2007), 560 pages, ISBN: 978-0321440303

Reviewed by Chris Oldwood

Highly recommended for Windows programmers

The first blog I discovered, and still read each day, is 'The Old New Thing' by Raymond Chen. He also writes a column called Windows Confidential which fills the final page in Microsoft's *TechNet* magazine. His book, which follows the name of his blog, is essentially a print version of more of the same – interesting historical facts and 'inside' information, surrounding the development of the Windows OS. There is a heavy bias towards GUI related issues as he works in the Windows Shell team, but there is also plenty of other low-level information such as in 'Understanding the consequences of `WAIT_ABANDONED`' which is relevant to any Windows programmer.

An operating system as old as Windows, which has had to transition from the 16-bit real world of DOS, through to multiple 32-bit platforms and then on to 64-bits is going to have more than just a few warts. On the face of it some of the APIs and behaviour may not make an awful lot of sense in today's largely 32-bit world and the obvious reaction of some detractors would be to blame it on poor design. The truth is often far murkier and frequently involves some nasty hangover from the 16-bit days – the entire sections devoted to `GlobalAlloc()` and `GetWindowText()` cover this ground thoroughly.

Microsoft takes backwards compatibility very seriously, and whether you agree or not with their sentiments, this book will show you some of the hurdles that need to be straddled for this to happen. For instance Raymond shows you 5 ways that developers can screw up implementing something at simple as `QueryInterface()`. He also describes badly implemented version checks and code that patches the OS, breaking it in the process. Couple this with the legal requirement of not being able to physically patch someone else's broken code and it's going to induce some creative (and entertaining) engineering.

There is another aspect to the book which I found puts software development into a new light, and that is the economies of scale that affect a company the size of Microsoft. To them a 1-in-a-million bug affects a significant number of customers, and they often have more beta testers than many products have actual customers. This can have particularly humorous outcomes such as in 'Windows brings out the Rorschach test in everyone' where he describes how various innocent images have managed to offend some users.

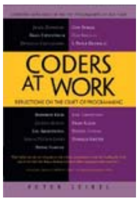
Although much of the grunge would be of more interest to developers coding to the C based API, there is still plenty of sound advice for those who live in the .Net world, and even some that transcends all OS's such as 'A cache with a bad policy is another name for a memory leak'. If I had to give my own definition of Schadenfreude it would probably contain a reference to this book.

Coders at Work

By Peter Seibel, published by Apress (2009), 632 pages, ISBN: 978-1430219484

Reviewed by Adam Petersen

Highly Recommended



Wow! This is an amazing book. I've been looking forward to *Coders at Work* since Peter published the first names on his blog two years ago. Given Peter's track-record , I knew he would do a terrific job, yet I'm positively surprised. *Coders at Work* is a book that I recommend, without any reservation, to anyone interested in programming or aspiring to become a programmer. It's that good.

The basic idea of interviewing legendary and influential programmers has been tried out before, recently in *Masterminds of Programming* (a book that unfortunately didn't live up to its promise). What sets *Coders at Work* apart from previous attempts is the quality of the interviews. Reading the book, it's obvious that Peter Seibel put a tremendous amount of effort into the preparations of the interviews. Peter knows the questions to ask each interviewee which results in deep and interesting discussions. Another sign that he succeeded are the heated discussions that emerged in the blogosphere immediately following the publication of the book. Because this book makes it clear that not even the experts can agree upon languages, type-systems, and methodologies.

Considering the interview subjects, it's an interesting mix. Some of the highlights for me were Joe Armstrong, Simon Peyton Jones, Peter Norvig, and Donald Knuth. And even if I've read several interviews with these people, in some cases visited talks given by them, Peter's interviews bring out a lot of interesting ideas and distilled programming wisdom that were new to me. Further, it was quite relieving to read about the low-tech approach to programming that most of them seem to share: debug through print-statements and a non-IDE approach.

Coders at Work is a book to read and think carefully about. What can I learn from the subjects? In what way do my approach to programming differ? For better or worse? No matter what, this book will help you improve. Along the way, enjoy Peter's writing and the highly interesting interviews.

ACCU Information

Membership news and committee reports

accu

View From The Chair

Jez Higgins
chair@accu.org

```
@t = (
"546865207965617206a75737420676f6e652077",
"6173206120707265747479207465727269666963",
"207965617220666f7220414343552e2020546865",
"20537072696e6720436f6e666572656e63652077",
"617320616e6f7468657220677265617420737563",
"636573732c206173207761732074686520417574",
"756d6e20536563757269747920436f6e66657265",
"6e63652e202052696320616e6420537465766520",
"686176652070757420746f67657468657220736f",
"6d65207465727269666963206973737565573206f",
"66204f7665726c6f616420616e64204356752e20",
"204f766572206f6e20616363752d67656e657261",
"6c20746865726520776173206120677265617420",
"6465616c206f66206c6976656c79206469736375",
"7373696f6e2028616e6420736f6d65206f662069",
"7420776173206576656e2075736566756c292e20",
"205468616e6b20796f7520746f2065766572796f",
"6e652c20616e642074686174276c6c2062652065",
"766572792041434355206d656d6265722c207768",
"6f20636f6e747269627574656420696e20776861",
"746576657220776179206f766572207468652070",
"61737420796561722e20",
);

foreach $t (@t) { print grep($_=pack("c",
          hex($_)),unpack("A2"x 20,$t)); }
```

Erratum

In the last issue of CVu (21.5) Paul Grenyer's article 'Java Dependency Management with Ivy' should have been labelled as 'Part 2'. Part 1 was in the prior edition (21.4).

Free tickets to SharePoint Techology and Enterprise Software Development Conferences

We have two free three day passports to the 2010 Spring SharePoint Technology Conference to give away to ACCU members. The conference is 10-12 February in San Francisco. We also have two free passports for the Enterprise Software Development Conference in San Mateo on March 1–3. The passports cover all sessions, receptions, and materials.

More information on the conferences is available at http://sptechcon.com/ and http://www.go-esdc.com/

Apply by email to ads@accu.org with your name, membership number, address, phonenumber, and company affiliation (needed for conference registration). The closing date for the SharePoint TC is midnight GMT 20 January, and for the ESDC is midnight 27 January. The winners will be notified on the 21 January and 28 January respectively.