Contents

Reports & Opinions

Dialogue

Features

Reviews

Copy Dates

C Vu 16.3: May 7th

C Vu 16.4: July 7th

Contact Information:

Editorial: James Dennett
914 24th Street,
San Diego
CA 92102, USA
`cvu@accu.org`

Advertising: Chris Lowe
`ads@accu.org`

Treasurer: Stewart Brodie
29 Campkin Road,
Cambridge, CB4 2NL
`treasurer@accu.org`

ACCU Chair: Ewan Milne
0117 942 7746
`chair@accu.org`

Secretary: Alan Bellingham
01763 248259
`secretary@accu.org`

Membership Secretary: David Hodge
01424 219 807
`membership@accu.org`

Cover Art: **Alan Lenton**
Repro: **Parchment (Oxford) Ltd**
Print: **Parchment (Oxford) Ltd**
Distribution: **Able Types (Oxford) Ltd**

Membership fees and how to join:

Basic (C Vu only): £15

Full (C Vu and Overload): £25

Corporate: £80

Students: half normal rate

ISDF fee (optional) to support Standards work: £21

There are 6 issues of each journal produced every year.

Join on the web at `www.accu.org` with a debit/credit card, T/Polo shirts available.

Want to use cheque and post - email `membership@accu.org` for an application form.

Any questions - just email `membership@accu.org`

Reports & Opinions

Editorial

Rather than talk about programming or programming issues, the work of WG21 etc, I've decided to talk about something far closer to my heart; the state of computer programming courses in education at all levels.

"Oh no, not another rant," I hear you say. Well, yes and no. Yes, it is – this sort of editorial usually ends up being such a thing. No it isn't, as instead of just complaining without any form or idea on solving the problem, I have attempted to do just that.

Combined with the fact that if something isn't done now, in ten years time we will be facing a potential disaster in terms of too many jobs available and not enough qualified programmers to fill them – a skills shortage that so far the computer industry has avoided by and large.

Recently, I undertook a small scale survey of colleges and universities in and around the North West of England as to how they perceived C Vu and what the ACCU actually represents. It came as a bit of a surprise how few organisations knew of us and moreover those who knew of us didn't really know what we did.

Then I started to think. Is it really that suprising?

Now what I'm going to say is not intended to be disrespectful and should not be thought of as a blanket for all.

Education and teaching of C and C++ in the UK is terrible. Some places are better than others, but from 4th year secondary school up to degree level, it is generally awful. Why is it awful though?

There are two reasons:

1 Lecturers not keeping up with the standards. I have known some establishments giving documents out for C++ which gives the end of its history as 1992.
2 Lack of investment.

They don't look it, but they are both part and parcel of the same thing.

Question why lecturers don't keep up with the standards and rely on old, broken code. Is it because they're lazy or just that they no longer care after many years of being over-worked? Have they lost interest and have just started to go through the motions? Is it that the students coming through from schools are that poor that they have to aim their material at such a low level that all they have left is despair?

Who is to say. What is clear is that coming through the education system are students who have been failed by the system. The upshot of that is that those going into the workforce, while they may be good, are more likely not to be. If they're not up to the task, then they won't be in employment for long and we (as a country and profession) end up with a shortage. Not a good prospect by any length.

I'm not going to get into the political debate over who is at fault ultimately, but I will say this. Having spent just about all of my working life in education (and recently moved into teaching), I can honestly say that the number of academic staff is too low and the hardware not up to scratch.

On average, it takes upto 8 months to write a course, have it approved and obtain funding. Academic staff are far from lazy. I spent around 8 to 12 hours on preparing a 2 hour lecture on the STL which is delivered once a week. For one academic university semester (12 weeks teaching), that amounts to a rough average of 100 hours of preparation time for 24 hours of lecture time. This excludes all marking time.

Now let's do the maths on that. Say you have 4 lecturers and 4 courses. That's 200 hours per lecturer per course – plus exam preparation, assessment preparation, marking, exam boards, assessment boards and all of the other related events they have to attend. What about these extras? An exam (if set by the academic rather than an exam board) takes around 40 hours to write and be approved. To mark 30 exam papers takes about 30 minutes per paper – 15 hours in total. Two assessments to write, say 10 hours each and then to mark them, an hour a shot. Exam boards and the likes, add another 5 days in total.

We have around 400 hours per course per member of staff – and that is without any teaching! It is very rare that a lecturer has only one course. If they have 3, we're talking 1200 hours – to put it another way, about 35 weeks of the year! Universities only effectively teach for 30 weeks a year, colleges for about 40 weeks a year. It is little wonder that courses are not updated.

With money as tight as it is, it is not unsuprising that lecturers are run off their feet – the first thing to go when money becomes tight is the lecturer – never anyone higher up. The establishments will have the same number of students, but suddenly the four lecturers become three. There is no time for development and no time for repair.

Combine that with a lack of investment in hardware (I know of one college which has working machines for the first 3 weeks, but as soon as they are put under stress, the 2Gb hard drives which have been in there for well over 4 years start to fail, so the machine fails) and you have a recipe for disaster. It gives students a negative opinion of the course, the college and moreover the languages.

Okay, let's see what happens in the schools. Back in the 1980s (when I was at school) we had two BBC micros. Both worked. Why two though when state schools were only supposed to have one? The school entered into a contract with a local company. The company provided the BBC micro in return for the company logo being embossed on the hardware supplied. A small price to pay and both the students (and staff) benefited.

Machines are only part of the equation though; appropriate lessons are the second part. Just teaching how to use Excel or Word is not a computer studies course which will prepare the student for further or higher education. That said, neither would a teaching a student extreme programming. A balance has to be struck.

Okay, that's the problem. Overworked, underpaid and under-resourced. How best can this be turned around?

Well, unless you know someone high up in government, it's unlikely that you will get education policy changed!

What can be done though is to sponsor lecturers and courses. If (say) Microsoft or IBM were to approach a number of colleges or universities and say "Look, we're willing to fund three academics for five years, but you have to teach these specific subjects," then the college/university would change to accommodate this sponsorship. Even if a company is upgrading machines and are disposing of (say) P3-450MHz machines, schools would willingly take them and in all likelihood be willing to have the company name splashed on the machines. Quite a few colleges would probably jump at those machines!

There is plenty which can be done by industry to help. If we return to the 1200 hour model per lecturer and suddenly move to a situation of six academics instead of four, the 1200 hours remains the same, but it is distributed over a larger number of staff. This means that the lecture courses can be updated; there would be no excuse for using out of date materials and then education would be far more accountable.

While we are not at the stage yet whereby there is a massive shortage, the number of qualified people coming out is far less. In 5 years time, the situation will be worse and 5 years after that – you get the idea.

What can the ACCU do?

One thing we will be starting in C Vu (with support on the ACCU website) is to run a beginners' C and C++ course. This is in response to comments from colleges and universities that the ACCU (and its magazines) are no longer relevant to those in education.

While we cannot sponsor lecturers and courses (we simply don't have the funds!), we will be happy to promote the activities of companies that do.

What can you do?

If you are in a position within a company to make this sort of decision, then make it. For the sake of the industry, make it. If you are not in that position of power, why not have a word with the higher up people and see if they have considered sponsorship at a local college or a university running a course appropriate to the business (for instance, Nintendo could sponsor a Computer & Video Games course and Novell something on server administration – you get the idea).

All right, enough of my ramblings and on with the show...

Paul Johnson

From The Chair

Ewan Milne <chair@accu.org>

The final preparations for the Spring Conference are being made as I write: the programme is packed with excellent sessions, bookings are streaming in, excitement is building. I'm confident that by the time you read this, we will be enjoying one of the best conferences yet.

Last issue James announced his intention to step aside as C Vu editor. We have had a volunteer come forward to take over the role, and an announcement can be expected at the AGM. The handover of the editorship, along with several other factors

Write for ACCU!

Pete Goodliffe `<pete@cthree.org>`

Your name in lights

ACCU membership is not just a magazine subscription; your subscription fee entitles you to more than just a few magazines a year. It's the opportunity to contribute, to see your name in lights, and to share your thoughts with the development community. In fact, if you don't then the ACCU journals will die.

We need members to write articles. And that means you! If you enjoy reading C Vu and Overload, and want to see them continue then please get writing something. You don't have to have been published before – new writers are positively welcomed.

Prizes! Prizes! Prizes!

Not only will you develop your own skills and have something excellent to put on your CV, you'll also stand the chance of winning one of the new ACCU awards. This year the ACCU is awarding prizes for published articles in a number of categories. These are:

- Best C Vu article
- Best Overload article
- Best article by a new writer

The awards will be announced at next year's AGM and the prize will be an ACCU T-shirt and untold acclaim. So what are you waiting for?

What to write?

We need articles at *all levels*, and on a wide range of topics. This means that you *already* have an article in you somewhere, something valuable that ACCU members want to read about. Really, you do. What are you doing right now? What do you know about?

Here is another selection of title suggestions. For more inspiration look at the lists in back issues.

- **.NET experiences**
- **1st steps with the STL**
- **Creating interfaces with Qt**
  Or any other toolkit
- **Writing mobile applications**
- **The tools I can't live without**

Don't hold back

Don't think you have nothing to write; you do. Don't think that your writing skills aren't up to it; try to sketch something out and you'll probably surprise yourself. We have a team of friendly people who'll help to craft your submission into the final printed copy, so you won't be on your own.

How to submit

You can send submissions by email to `editor@accu.org`. Plain text is perfectly acceptable; there is a Word document template you may wish to use if you want to retain formatting. That's all there is to it – ***get writing***.

currently abroad, make this a good time to consider the future direction of the ACCU journals. As you know, the association has for the past ten years published two journals: Overload as well as C Vu. Overload began as the journal of the C++ Special Interest Group, and has grown to be the home of longer, more in-depth features than those found in C Vu, still the journal of the association itself. Both publications serve very valuable purposes, and we'd like to think both make highly informative and entertaining reading. This is, of course, due to the hard work of a dedicated team of members. The job is not easy however, and recently gathering material for C Vu has been a particular challenge.

We now need to consider the best way of continuing the high quality we all enjoy from the journals, and evolving them to best reflect the association in its current form. The success of Overload has led to 90% of the membership taking both magazines. Given that production and distribution of the journals is our largest running cost, there is certainly a practical motive for considering merging the two into one. But would the result be greater or lesser than the sum of its parts? At least, if we want to keep the two separate, the process for C Vu requires review: at present too much work is done by a single editor (at this point I should also mention the invaluable contribution of Pippa, the production editor). Overload's editorial panel has proved to be a very useful way of spreading the workload. I feel that a similar panel would be a good idea for C Vu. It could perhaps even be the same one, or at least share some members, in order to maintain an editorial coherence between the journals.

There is no doubt that these are major issues, and careful consideration of them is needed. For that reason there is an item on the agenda of the AGM to discuss them. I look forward to hearing your views.

Membership Report

David Hodge `<membership@accu.org>`

Have a look at your entry in the handbook. If it says 'details withheld' do you want this to continue when the next issue is produced in May? Just email me and I will get it changed. Also don't forget to email me if you change your postal or email address. Our current membership stands at 1047 with 141 new members this year but still making us 78 down on last year. There are 134 members paying by standing order, if you would like to join them just send me an email. Looking at the first 2 letters of the postcode in the UK the highest concentration of members is OX 51, RG 30, GU 27, SO 21, BS 19, KT 18, SW 18.

Standards Report

Lois Goldthwaite `<standards@accu.org>`

This month's report comes from the Ecma TG5 meeting in Melbourne, Australia. This is the group which is developing a "binding" for C++ to the CLI (Common Language Infrastructure) environment. CLI is the standardese name for what Microsoft is commercialising as .Net. Four members of the UK C++ panel are sitting in as liaisons from the ISO C++ standards committee, WG21, though one of them, Jon Jagger, also has another hat to wear as convenor of TG2, the Ecma C# committee.

This is the first time ever that such a liaison relationship has been established between working groups in Ecma and ISO, and perhaps it will lead to more such cooperative efforts in future.

The announced intent of TG5 is to maintain complete compatibility with standard C++, while also adding extensions which will make C++ a leading platform for CLI development. However, these extensions are extensive, and already pressure is building for WG21 to adopt some of them into ISO standard C++. To make C++/CLI support two divergent programming philosophies at the same time leads to severe complications for compiler writers, such as a dozen or more new context-dependent keywords – they are only parsed as keywords if used in certain specific ways; otherwise they may appear as variable or type names in a program. Real C++ keywords, of course, are not valid as user-defined identifiers.

Not breaking legacy code is a worthy goal, but I fear that this extension is one example of how C++/CLI will make it harder to teach and learn C++: "Here is a list of keywords which convey information to the compiler ... your program won't compile if you try to use these as identifiers, except for a dozen or so which are OK."

Another way in which the proposed extensions will confuse novice programmers is that the rules for defining classes will become much more complex, depending on which of several new flavours of class is involved. These complicated new rules affect whether you need to write such words as "public" and "virtual" when defining a hierarchy of classes.

TG5 is addressing issues which are important for the future of C++, such as better specification for mixed-language programming. And some – though definitely not all – of the new features would be useful in standard C++, and some of the others would be harmless and should be adopted for compatibility. But I would feel better if TG5 just stated outright that they are producing not just a binding with some extensions but a new dialect beyond C++, and jettisoned the complicated workarounds and even kludges that are being adopted to maintain simultaneous support for both environments. Both standard C++ and C++/CLI would benefit.

Looking briefly at the TG2 meeting earlier in the week, some new features are being adopted into C#. These include generics (which have some of the benefits of templates without the thornier complications), iterators (a simpler syntax for enumerating the contents of a collection), anonymous methods (you can write the source code for a delegate function inline inside a block scope at the point of declaration, without having to assign a name and other function overheads), partial types (a class definition can be reopened and assembled in several pieces – this is intended to make it easier to generate code mechanically), and static classes, which cannot be instantiated but contain static member functions (a workaround for the lack of stand-alone functions).

If you are interested in participating in the UK panels for language standards, please write to `standards@accu.org`.

Dialogue

Student Code Critique Competition 27

Set and collated by Francis Glassborow
Prizes provided by Blackwells Bookshops & Addison-Wesley

Please note that participation in this competition is open to all members. The title reflects the fact that the code used is normally provided by a student as part of their course work.

This item is part of the Dialogue section of C Vu, which is intended to designate it as an item where reader interaction is particularly important. Readers' comments and criticisms of published entries are always welcome.

Before We Start

This is the last column under my beeline as setter and collator. David Caabeiro responded to my appeal for a volunteer to take over from me. Submissions and other material should be sent to him at `caabeiro@vodafone.es`. He is keen that code critiques should expand beyond the purely C and C++ ones I have provided. With that in mind any readers who spot an apt piece of code in Java, Python or even C# should send it to him. The intention is that there will always be a C or C++ piece of code but where available there will be a code sample from one of the other languages.

Another change is that I will keep contact with this column by submitting a 'Teacher's Critique' of C and C++ student code. In other words alongside the competition entries there will be an extra critique from me which will usually include the solution that I would have provided (which will also, of course, be open to question and refinement).

I hope that everyone will give David plenty of support because there is nothing so depressing as lack of response. In the virtual classroom we cannot see the looks of enlightenment, puzzlement or boredom on other faces so we have to rely on the 'students' making their participation visible.

Student Code Critique 26 Entries

This time the problem is still about some student code, however it is code that works, but how do you help the student to move forward? Yes, I know the simple answer but I am looking for something more.

The following program (below my question) produces the results I want, but I am trying to create a `for`*-loop to replace all of the* `cout` *statements.*

What I have so far is:

```
for(int i = 0; i <= len; i++)
  cout << str[i + 1];
```

This will produce: "eed help with C++."

Could someone clue me in on how to create the `for`*-loop? Should I be using a nested for loop? Any help would be appreciated.*

Student's Program:

```
#include<iostream>
#include<cstring>

using namespace std;

void main (){
  const int arraySize = 20;
  char str[arraySize] = "Need help with C++.";
  int len = strlen(str);

  cout << "The sentence \"Need help with
                                  C++.\" has "
     << len << " characters."
     << endl << endl;
  cout << str << endl;
  cout << str + 1 << endl;
  cout << str + 2 << endl;
  cout << str + 3 << endl;
// 13 similar statements snipped
  cout << str + 16 << endl;
  cout << str + 17 << endl;
  cout << str + 18 << endl;
```

From Walter Milner

Dear Student

I think I should remind you that the set text for this course is *Accelerated C++* by Koenig and Moo, and you are supposed to read it. I looks to me as if you haven't.

First of all let's look at what you have. We can see more clearly what is happening if we also display the index, together with the data as characters and code, as

```
for(int i=0; i<=len; i++)
  cout << i << " " << str[i+1]
       << " " << (int)str[i+1] << endl;
```

which on a machine using ASCII yields -

```
0 e 101
1 e 101
2 d 100
3   32
4 h 104
{similar lines snipped by FG}
17 . 46
18   0
19 + -64
```

So your loop outputs 20 characters (0 to 19 inclusive) The 19th is the zero marking the end of the string, and the 20th is whatever is in memory after that. We can simply fix your loop as

```
for(int i=0; i<len; i++)
  cout << str[i] << endl;
```

However there are some issues with the way you are doing this. Firstly, how did you choose the value of `arraySize`? Did you count the letters in the string and add 1 for the 0 at the end? Pretty tedious, and it makes using `strlen` pretty pointless.

Secondly, the approach is not easy to adapt to other strings – in other words there is a close link between the algorithm and the actual data it processes. To cope with another string, would you have changed `arraySize`? To what? You could fix it at 1024 and limit this to strings of length less than that, but that would be pretty unsatisfactory.

Using an array of characters is a common way to represent a string in C. Alternatively you might have said

```
char * str = "Need help with C++.";
```

Then said something like

```
for(char * ptr = str; *ptr; ptr++)
  cout << *ptr << endl;
```

which relies on the fact that at the end of the string `*ptr` is 0 and therefore false, ending the loop. In other words it depends a lot on the way the string is stored – the details of the implementation. This is how it is done in C – but we're supposed to be using C++.

Why have you been given this as a task anyway? What is the point of outputting a sentence one letter at a time? Hopefully you realise that this is an exercise concerned with the two basic questions:

1. How do we have something of interest in a computer?
2. How do we do things with it?

In this exercise the thing of interest is an English sentence. We need a way of representing that inside the machine. The second question in this case is how to 'go through' that representation – to traverse it. This is a very common real task. You traverse representations to search them, count them, add them up, change them and so on.

We want to do this with the minimum of effort, and the maximum of generality – so in future we can do it with even less effort. If possible we want to avoid the **details** of how it is done – we just want to think of it as a string, not as bytes or values in RAM. In other words we want to work at as **abstract** a level as possible.

But the Standard C++ Library has such a thing, not surprisingly called `string` (surely you got as far as Chapter 1 in *Accelerated C++*?). We just say:

```
string str = "Need help with C++.";
cout << "The sentence \" << str
     << "\" has " << s.length()
     << " characters." << endl << endl;
```

How is that string stored in memory? Who cares? Not our problem. We want to traverse that string from start to end, so we can just say

```
for(string::const_iterator i=s.begin();
        i!=s.end(); i++)
  cout << *i << endl;
```

If you've lost the handout on iterators, see chapter 5 in *Accelerated C++*.

[As a teacher I would have to say that a struggling student would almost certainly give up the course if faced with such a critique. Of course the work might be that of a bright but lazy student but it does not have the hallmarks of such work. To me it looks like the work of a very conscientious student who has completely failed to grasp a couple of important elements, such as how for-loops work. Francis]

From Colin Hersom <colin@hedgehog.cix.co.uk>

Le us solve the student's immediate problem of how to turn a list of similar-looking statements into a `for`-loop to save repetition.

The existing code looks like:

```
cout << str << endl;
cout << str + 1 << endl;
cout << str + 2 << endl;
cout << str + 3 << endl;
etc.
```

Taking this very slowly, the first thing to do is to ensure that all the statements have a similar form, so I shall change to first line to:

```
cout << str + 0 << endl;
```

This gives them all an addition operator, but adding zero has no other effect. Now we can progress to producing a list of statements that are all textually identical by introducing a variable to replace the constants in each line:

```
int i = 0;
cout << str + i << endl;
i++;
cout << str + i << endl;
i++;
cout << str + i << endl;
i++;
..etc
```

Now that really can be replaced by a loop - provided that we know when to stop. The final statement in the original code is

```
cout << str + 18 << endl;
```

so we need to stop when `i` passes 18, i.e. continue while `i` is less than 19.

We can now wrap up all those identical statements into this:

```
int i = 0;
while(i < 19) {
  cout << str + i << endl;
  i++;
}
```

and we realise that the construct:

```
iterator = start;
while(condition) {
  body-statements
  iterator++;
}
```

can be replaced by a `for`-loop:

```
for(iterator = start; condition; iterator++)
  body-statements
```

which gives us our `for`-loop:

```
for (int i=0; i<19; i++)
  cout << str + i << endl;
```

"Magic numbers", i.e. constants used in their numeric form, should be avoided, except for zero in most cases. Sometimes these constants are part of the problem domain, e.g. the acceleration due to gravity, in which case they should be provided by a named constant (or a macro in older C implementations). Here, the 19 output statements are due, I assume, to the length of the test string. Since we have calculated the length, we can use it. This also means that if the test string is changed, the number of lines exactly equals the length:

```
for (int i=0; i<len; i++)
  cout << str + i << endl;
```

Some people would argue that the pre-increment `++i` should be used in place of the post-increment here. For integers it makes no difference. For more complex iterators it might do, so it is possibly better to stick to a consistent style to save surprises later.

If we go back to the beginning of the student's code, we spot that the main routine has been declared to return `void`. This is incorrect – it always returns `int`. My compiler, with warnings enabled, tells me this and kindly(?) changes it to an integer return.

Then comes another "magic" constant. Why choose 20? Ah, it must be that string being used to test again. This is problematic because if the string were changed to "I really need a lot of help with C++." then that array of 20 characters is not going to be big enough and all sorts of nasty things might happen. C++ has inherited from C the ability to dynamically size arrays from their initialisation lists, and also to initialise an array of `char`s by a string, so we can use this to create enough space automatically:

```
char str[] = "Need help with C++.";
```

We then wonder why we should rely on the function `strlen` to find the length. The compiler has already does this for us to allocate the size of the array "`str`", so we can use the `sizeof` operator to get the value at compile time, remembering that the `sizeof` operator will include the trailing null, whereas `strlen` does not. If we put this into a constant then the compiler should use the literal value rather than a variable:

```
int const len = sizeof(str) - 1;
```

The only proviso here is that if the literal string contained embedded nulls then `strlen` would only count to the first such null, whereas `sizeof` uses the real number of characters in the string.

The whole program now looks like this:

```
#include <iostream>
using namespace std;

int main() {
  char str[] = "Need help with C++.";
  int const len = sizeof(str) - 1;
            // Ignore trailing null
  cout << "The sentence \"" << str
       << "\" has " << len
       << " characters" << endl << endl;
  for(int i=0; i<len; ++i)
    cout << str + i << endl;
  return 0;
}
```

Note that I have also removed the duplication of the string embedded in the introductory output statement. That ensures that if we change the test string then the output is consistent. Also the header `cstring` is no longer required to declare `strlen`.

Although I would prefer not to include the whole of the `std` namespace and instead specify specifically which classes or functions I want, my compiler includes the namespace as standard and so I cannot check for errors if I omit some declarations. Therefore I have left this statement in place.

OK, so that has fixed the initial problem, of removing the duplicate `cout` statements. The student said that he wanted to write the body of the loop as:

```
cout << str[i+1];
```

for some value of `i`. This statement only puts one character from the string onto the output and there is no `endl` either. This indicates that the requirement was to print out individual characters. There is rarely a need to do this on a string, in real life, but as an exercise it is useful to know how to access parts of the string, so I assume that this is what is required. The student also mentions nested `for`-loops, which also suggests that individual character writing is wanted.

I wonder whether the requirement was stated as "replace the `cout` statements with a loop?", meaning "replace all `cout` statements with one inside a loop" while the student understood it to be "replace each `cout` statement with a loop of its own". Such a misunderstanding would not be unusual.

Let us assume that the student's interpretation is correct, how do we go about this? Each iteration in the loop, in its current form, prints the string starting from the `i`'th element. The way that `char*` strings are handled (also an inheritance from C) means that all characters from the pointed-to one are printed, until a null character is found, which is the same as saying until the string length as defined by `strlen` is reached.
The inner loop plus the end-of-line should produce exactly what the

```
cout << str+i << endl
```

line produced earlier, i.e. in pseudo-code:

```
for all characters from the i'th to the len'th
  print the character
print end-of-line
```

Using the conventional counter name `j` (when the outer counter is `i`), we get:

```
for(int j=i; j<len; ++j)
  cout << str[j];
cout << endl;
```

Putting this inside the existing loop, I get:

```
for(int i=0; i<len; ++i) {
  for(int j=i; j < len; ++j)
    cout << str[j];
  cout << endl;
}
```

The `char*` (and `char` array) style strings are very much a legacy from C and their use should really be discouraged when teaching C++ to new students, who should be introduced to the `std::string` class instead. This student appears to be trying to learn about loops rather than strings and so an array of integers might have been more appropriate. Again, a `std::vector` might be even better, since this provides more safety than a plain C array.

On the other hand, the student may be in an environment where they are using C++ as "a better C" and steer clear of the more complex aspects of C++. The sample code is very much along these lines, since it only uses `cout` and the stream insertion operator from C++, all the rest is plain C. From a C-programmer's point of view, using C++ in this way is a natural progression – I would even own up to doing it myself – but it does mean that some C ways of doing things need to be re-learnt to use C++ to its fullest power. If the student is in this sort of environment, then learning better C++ techniques will do no harm, and may well lead the student to become the local expert on C++. Helping others with their C++ will enhance the knowledge of the teacher too, as Francis keeps trying to tell us.

The Winner of SCC 26

The editor's choice is:

Colin Hersom

Please email `francis@robinton.demon.co.uk` to arrange for your prize.

Student Code Critique 27

(Submissions to `caabeiro@vodafone.es` by May 10th)

There are at least two issues with this issue's problem code. The first is finding the immediate logic error in the code that results in false positives. The second, to my mind more important issue, is the student's approach to solving the problem. It is relatively easy to learn to write syntactically correct code but that is not programming any more than writing grammatically correct English is writing poetry or a novel.

Please submit a critique of the code as it is and then append a critique of the student's approach to the problem s/he was set.

In this exercise an ugly number is defined as a compound number whose only prime factors are 2, 3 or 5. Note that a factor can repeat so 20 (2*2*5) is an ugly number. The following program outputs ugly numbers correctly, but also outputs prime numbers. I can't understand why this is happening because the pointer returned by `pf()` points to a zeroed 1st element of the array `flist` when the input to `pf()` is prime. It should fail the `while()` test and the next number should be factored. I can kludge this with a separate test for primes, but I would like to understand why it's not working as is.

```
/* find "ugly numbers": their prime factors
    are all 2, 3, or 5 */
#include <stdio.h>
#include <stdlib.h>
#define SIZE 20 /* max number of prime factors */
#define TWO 2
#define THREE 3
#define FIVE 5
int *pf(int number);

int main(int argc, char *argv[]) {
  int *flist,idx, n, ugly = 0;
  int start, stop;
  if(argc != 3)exit(EXIT_FAILURE);
  start = atoi(argv[1]); /*enter a range */
  stop = atoi(argv[2]);
  for(n = start; n < stop +1; ++n) {
    idx = 0;
    flist = pf(n);
    while(flist[idx]) {
      if(flist[idx]==TWO ||flist[idx]==THREE
           ||flist[idx]==FIVE) {
        ugly = 1;
        ++idx;
      }
      else {
        ugly = 0;
        ++idx;
      }
    }
    if(ugly == 1)
    printf("%d\n", n);
    free(flist);
  }
  return  0;
}
/* find the prime factors of number */
int *pf(int number) {
  int *flist, quotient=0, divisor=2, idx=0;
  flist = calloc(SIZE,  sizeof(int));
  while(divisor < number + 1) {
    if(divisor == number){
      flist[idx] = quotient;
  /* add last factor to list */
      break;
    }
    if(number % divisor == 0) {
      flist[idx++] = divisor;
      quotient = number/divisor;
      number = quotient;
    }
    else ++divisor;
  }
  return flist;
}
```

Francis' Scribbles

Francis Glassborow `<francis.glassborow@ntlworld.com>`

Repository of Projects

ACCU has over a thousand paid up members and its periodicals are read by far more than that so I am very disappointed with the response to my request for help in the last issue of C Vu. Almost every reader is some kind of programmer (not quite all because I know that C Vu is also read by quite a few hangers on who enjoy the non-technical or at least the non-programming part.). Almost every programmer has learnt his or her trade by actually writing programs to solve problems. The lucky ones actually got to write programs that did something they found interesting or useful.

It takes about ten minutes to write up a simple description of a potential programming task. If each of you just added a single project, preferably one you would find interesting, we would soon have a real resource to help those aspiring to become skilled programmers.

I won't bore those of you familiar with the story of 'Stone Soup' but if you do not know it take a moment to visit `http://spanky.triumf.ca/www/fractint/stone_soup.html`. Some activities need a well-defined team working together but many other things can be done piecemeal with each individual adding a little bit. If you were bored by the assignments you were given when learning to program see if you can offer the next generation something better. On the other hand if you had a couple of thrilling assignments that brought programming to life for you, pass them on to the next generation.

Another way to add to the resource is to provide a link to a publicly available source of programming projects/assignments. I might archive a copy (with permission of the owner) in case it eventually goes away but I would be very happy to just supply a link from my Project Repository site. Whatever you do, don't do 'nothing'.

That neatly leads me to the following.

A Simple Program

Look at the following code and decide what is wrong with it. When you have done so, decide what it has to do with the previous section in particular and this section of C Vu in general – well actually the whole of your life.

```
#include <iostream>
#include <string>
#include <vector>
using namespace std;

int main() {
  try {
    string input;
    vector<string> storage;
    while(true) {
      getline(cin, input);
      storage.push_back(input);
      if(!cin) cin.clear();
    }
  }
  catch(...) { cout << "I died\n"; }
}
```

It does not take very long to determine that the above program lacks an essential ingredient that we require of a program. It has the form of a program, it will compile and it will even execute. In doing so it will absorb all the input until one day it is full up and falls over dead.

Now this column turns up in a section of C Vu entitled 'Dialogue' which is meant to be a strong hint that there should be output from the reader as well as input. Reader and writer are supposed to interact.

If you think this is something similar to what I have written before (Hey, that is an idea James, how about me just repeating any section of my column which gets no response on the grounds that it obviously never reached intelligent processing.) my question is: 'What did you do about my moaning last time?'

More on Spam

Well since I last wrote on the subject my ISP started filtering spam, or claims to do so. Now what kind of email is one you do not ever want to receive? What kind of email would you never willingly send? So what kind of email is often not being rejected by ISPs?

Yesterday I received 270 emails. 102 of those contained one or other of the more recent viruses. My anti-virus software detected the problem, quarantined the offending item and then sent me a message telling me it had done so (which message I promptly told my mail reader to reject, but that is entirely internal to my system). Note that anti-virus software pretty quickly starts identifying new viruses as long as you keep it up to date.

113 of yesterday's emails were messages from elsewhere telling me that some item purporting to come from me (though actually nothing to do with me) had been rejected. Most of those rejections were because the email in question had a virus-infected attachment. Checking back to those 102 emails of the last paragraph revealed that a couple of dozen of the virus laden emails were actually such rejections (or at least purported to be).

Much of the rest of my email was plain ordinary spam that had managed to avoid my ISP's filter.

What I want to know is why any ISP is accepting email with a detectable virus in it for onward transmission. To my mind anyone whose email contains a detected virus should immediately have their account suspended. Note that an ISP has to be able to identify the origin of email from one of their clients. They know which client sent it. Of course address forgery happens but to send email I need a physical connection of some sort and it is that connection that needs to be snipped as soon as my machines start attempting to mail out virus-laden software.

So do you know of an ISP that both filters out all email containing an identifiable virus and suspends the account of any client who sends such an email? Because if you do I would seriously consider moving my business to them.

You will have noted that I continued to qualify 'virus' with identifiable. Of course there will be new viruses that will remain undetected for a few hours but as soon as detection is possible an ISP should apply it and destroy (not return) all infected email as well as suspend clients who originate such email. Note that the suspension is also beneficial to the client because they will know very quickly that something nasty got through their defences.

Personal Data

I was recently attempting to register a product that I had bought. The registration required me to provide an email address even though there was no valid reason for the supplier to have that address. They had not even bothered to create some ersatz reason such as sending me a logon code. I think it is time that we all started digging in our heels when people ask us to supply personal data. They need to tell us why and for what it will be used.

Now it is hard to argue about such things when you need an updated driver so that your purchase can be used on something such as Windows XP. You need the update and they will not provide it until you hand out personal data. Isn't this demanding 'personal data' with 'menaces'? I.e. isn't it a form of extortion? Should we not identify this as a place where the Data Protection Act (or whatever is the equivalent in your country) is tightened up? Something along the lines of a substantial fine for asking for personal data without informing the person as to how it will be used.

It would probably need a revision of the 'Sale of Goods Act' (or equivalent) to make it illegal to withhold software upgrades on the basis that the customer does not want to provide personal data. Yes, of course I can return the goods, but when was the last time you did that on the basis that you could not use them unless you gave the manufacturer your telephone number?

Computers greatly increase the threat to our privacy because they allow mining of immense databases. We need to fight back by limiting what we are 'required' to provide to those databases. Telling someone they cannot use their newly purchased scanner till they hand over their email address is a 'requirement' in all but name.

Two Little Tools

In the days when I edited C Vu I tried to get readers to share their software experiences. With a couple of exceptions that was a dismal failure. Let me share my own experiences with you.

As a result of having to learn enough about HTML, CSS and the like in order to provide a website to support my book I found myself looking around for some reasonable tools both for preparing the files and for validating them. I have no doubt that there are many such products available but all I can tell you about is the ones that I eventually found, tried out and then purchased.

The first one is a product called 'Note Tab' which is intended to be a replacement for Windows' Notepad. I very soon realised after getting the free version from `www.notetab.com` that this was an immensely more powerful tool than the one it was replacing. Not only did it handle multiple open files but allowed me to select what sort of thing I was working on. I could select HTML and get a bundle of useful extras to work on HTML source. I could then select CSS and get another bundle of tools for working on a cascading style sheet. And so on. I could even add my own customised support for some other activity.

I was impressed enough that I skipped their 30-day free trial of the latest version and simply signed up for the full professional version for $19.95 + VAT (which is now levied on electronic purchases based on the origin of your credit card). The weakness of the dollar made that pretty painless.

I highly recommend this product, and if you use a MS Windows based machine try it out. If you know of something better please take a few minutes to tell me (and through me, the readers of C Vu) about it.

The second issue with preparing web pages is having some reasonably adequate validator. The one provided by W3C is OK but it doesn't help very much if something simply fails to pass. What I needed was a tool that not only validated my HTML (actually, XHTML transitional) but would give me some guidance about errors and some advice with regard to warnings. *[When a page fails at the W3C validator, the reasons for the failure are given on te results page with quite a comprehensive set of reasons for the failure - PFJ]*

A search of the web came up with 'CSE HTML Validator'. Details can be found at `www.htmlvalidator.com`. This is another product that has a free version which in itself is a helpful tool, or at least I found it so. I went on to try the full standard version. At this point I discovered a further ability of Note Tab, it can call out to other tools and one of those is CSE HTML Validator. I could work on a web page and hit a single hotkey to get it validated (sort of like compiling source code).

This product is rather more expensive but at $69 for the full standard version it does not seem unreasonable.

Here again you may know of something better or simply an alternative. Please share your experiences. In the meantime, if you are maintaining a website, consider trying the lite version of this tool and see how you like it.

A Big Tool

The previous two programs were useful but with limited purpose. The next product is rather more substantial. I am preparing to write my next book and need a reasonable IDE that will run on both MS Windows machines and Linux ones. Everything I tried had serious flaws, like being awkward with console style programming because the developers assumed that everyone would want to write full GUI programs. Or they expected a bundle of overheads such as requiring wxWindows for graphics programs. *[wxWindows recently changed its name to wxWidgets due to a potential problem with the original name - PFJ]* I really do not want my readers learning more than one thing at a time. Learning to write good C++ is enough without having to struggle with the eccentricities of a complex graphics library or an all singing GUI support library.

I kept looking and a few days ago I came across MinGWStudio. The MS Windows version certainly looks good and has a quality feel to it. Of course it is early days and it will have to be stress tested before being let near my readers. There is also the issue as to whether the Linux version will also pan out OK. Ian Bruntlett is preparing to test that with my library source code. If that works OK then it looks like another product whose praises I will be willing to sing.

It is free but the developer is asking for donations, which considering the magnitude of the product seems very fair to me.

If you already know about this product please share your experiences. If you do not I think it might be worth your trying it. You can get more details at `www.parinya.ca`. By the way it also provides a FreeBSD version, if anyone reading this uses that OS and would like to try the Unix version of my library with it please drop me a line.

Australia

By the time you read this I will have got back. I hope that my absence for three weeks has not caused anyone any problems. I will let you know what gets done out there in the next issue. However as a result of my last column I got an email from a long-standing ACCU member who now lives in Melbourne. I knew Alec Clews when he lived and worked in England and I am very much looking forward to meeting him again. I am glad I told you all where I was going otherwise I would have missed meeting Alec. Friendships that endure over the years and over long distances are made so much easier by modern technology. Do not get conned by those who claim that computers turns us into recluses; they can but they do not have to.

Commentary on Problem 13

Look at the following C code. Why does `qsort()` fail? Please note that the `compare()` function does rank any two objects of type `X`.

```
struct X {
  int i;
  int j;
  int k;
};
int compare(void * p1, void * p2) {
  struct X * x = p1;
  struct X * y = p2;
  if(x.i > y.i) return 1;
  if(x.j > y.j) return 1;
  if(x.k > y.k) return 1;
  return x.i + x.j + x.k - y.i - y.j - y.k;
}
int main() {
  struct X array[10];
  /* code initialising array */
  qsort(array, sixeof(X), 10, compare);
  /* etc. */
  return 0;
}
```

One of the key assumptions for sorting is that there is a well-defined final order excluding that items that compare equal may be re-ordered from one invocation of a sort algorithm to another. The problem with the above `compare()` is that there is no such guarantee. It is perfectly possible to have three `X` items `a`, `b`, `c` such that `a>b`, `b>c`, `c>a`. That breaks the fundamental requirement for sorting.

This reminds me of all those other cases where there is no ordering. We assume that x is better than y, y is better than z implies that x is better than z. That is a fundamental flaw in our thinking. As soon as our choice is based on more than one criterion the assumption is false. To put it another way, there is often no best choice and we have to make value judgements as to what is more important to us.

Good team captains implicitly understand that 'better' is not an absolute when they seek to exploit their opponents' weaknesses. For example, if your cricket opponent does not have any leg spin bowlers you can safely select a batsman who is vulnerable to leg spin.

Cryptic

Christmas Competition

Well there were no other entries other than those from John Kewley and Richard Blundell. I see no reason to distinguish between their excellent clues so they each win a copy of any book from the C++ In Depth Series or a copy of either the C or the C++ Standard as published by Wiley.

If they will email me their choices I will get copies out to them as soon as I have recovered from my visit to Australia.

Time For

Another form of numerical clue is to use the time or date of some event that is well known to the readership. Clues actually using a time tend to be very much local to groups but dates can be more general:

When next Julian has one that Gregory skips. (4 digits)

If you saw that clue without any context you might not immediately think of Gregorian and Julian calendars. However the clue does hint at time by use of 'When next' and so the experienced solver of cryptic clues should stop and think about calendars. What is the difference between the Julian and the Gregorian calendar? The algorithm for determining leap years. And now the answer should be clear. 2100 AD is the next time that the Julian calendar inserts a leap day when the Gregorian one does not.

On Reflection

Some of the digits have reflective symmetry (0, 1, 3, 8) though '3' only reflects vertically. Note that 0, 1, 8 also have rotational symmetry. '6' and '9' are special because they are rotations of each other.

This property of the glyphs we use to represent digits opens up some potential for cryptic clues:

The beast turning round causes a very British call for help.

Gives '999' to anyone who either knows their Bible or the English emergency phone number.

Now see if you can come up with a good clue for '9901'. Of course you can use any of the ideas I have presented over the last year as well as any of your own.

Francis

Features

Sponsored Feature

C++/CLI, Ecma TC39/TG5, and SC22/WG21

Thomas Plum `<tplum@plumhall.com>`

There have been many languages for writing applications, but relatively few foundation platforms which support applications that are written in various different languages. We've had assembler (proprietary), then C (an ISO standard), and now we have the Common Language Infrastructure (CLI). The CLI standard is ISO/IEC 23271; the same content is also available online at `http://www.ecma-international.org/publications/standards/Ecma-335.htm`.

For years, C++ has been the most popular language for applications and middleware, but it has usually been hosted on a C base library. Very seldom is a C++ base library made available for use by other diverse languages. For example, libraries that make extensive use of templates (like the standard C++ library does) are difficult or impossible to be used by languages other than C++.

The important features of CLI as a base library have historical roots in a variety of previous technologies:

- Robust (strongly typesafe, like Ada)
- Inheritance (single, like Smalltalk)
- Garbage collected (like Smalltalk)
- Hardware-oriented datatypes (like C)
- Twos-complement (8-16-32-64, like all modern PCs and workstations)
- Exception-handling (like Ada and C++)
- Object (single root of all inheritance, like Smalltalk)
- Universal Intermediate Language, or "IL" (with JIT compiler, like ANDF)
- Shared libraries available to all languages (like C)

CLI provides a base which can support applications written in ECMAScript (Javascript, Jscript), COBOL, Perl, Java, C#, etc., as well as C++.

CLI types are allocated into the CLI heap, using accurate garbage-collection ("GC"). In C++ these are the "managed types". Microsoft's first extensions of C++ for CLI were called "managed C++". Programmers complained about the awkward "bolt-on" feel and asked for a smoother integration. The new integrated design is known as C++/CLI. Microsoft offered the C++/CLI spec to Ecma International for standardization within the Ecma process. Ecma Technical Committee 39 (TC39) is the group which has already standardized ECMAScript, C#, and CLI, and is currently standardizing Eiffel. Within TC39, Task Group 5 (TG5) is standardizing C++/CLI; the author is convener of TG5.

Here are a few comparisons between development within Ecma versus the national-body process within ISO/IEC JTC 1. The members of Ecma are companies; they include Apple, HP, Intel, IBM, and Microsoft. Ecma is supported by one fairly large annual fee per member company, and its standards are then given away for free via internet or CD-ROM. (Non-profit organisations such as universities and government laboratories can join Ecma for free.) The members of ISO, IEC, JTC 1, and SC22 are national standards bodies. National-body development is typically supported by a mixture of fees for documents and fees for individual participation. Development within Ecma is usually quicker than development within JTC 1. The original CLI and C# standards were developed in about one year, using monthly face-to-face meetings with a teleconference midway between each face-to-face. The schedule of TG5 is somewhat less accelerated; our meetings are spaced about six weeks apart, with occasional teleconferences.

TG5 met first in College Station (Texas), then in Kona (Hawaii); we will meet early March in Melbourne (Australia). Participant companies include Microsoft, IBM, Plum Hall, EDG, and Dinkumware. Invited experts have included Bjarne Stroustrup, Gabriel Dos Reis, and Jon Jagger. TG5 has established liaison with SC22/WG21, whereby members of WG21 national-body C++ panels can participate in a TG5 reflector, receive TG5 documents, and attend TG5 meetings. Since WG21 is actively engaged in the next revision of the ISO C++ standard ("C++0x"), TG5 wants to avoid pre-empting or conflicting with that ongoing revision; the liaison process provides feedback to TG5 in this regard.

C++/CLI tries to avoid breaking or changing the behaviour of already-existing C++ code. We have used creative methods for giving the C++/CLI programmer a first-class ease-of-use while minimizing (perhaps eliminating) clashes with existing user-space identifiers. As of February 2004, the C++/CLI draft still clashes with one identifier, namely the keyword `gcnew`. In C# any name can be prefixed with "`@`" to indicate "I'm not a reserved word, I'm an identifier", such as `@gcnew`. This allows the use of CLI fields, enums, etc., written in various other languages which don't use the same keywords. Whether C++/CLI will use the "`@`" like C#, or a different syntax, is still being decided.

C++ has always been a multi-paradigm language, and C++/CLI will add even more choices. C++/CLI can be used as a producer language to create CLI base libraries and middleware which can be used by the full range of CLI consumer languages. In this paradigm, we aim to equal the expressiveness and efficiency of C#. But C++/CLI can also be used as a consumer language for CLI applications, using the full power of native C++ and linking with any APIs that are currently available to native C++. Even in the latter case, the C++/CLI software can still publish CLI APIs to any other CLI language. (But the native C++ parts are likely to be unverifiable, and therefore unsuitable for use in security-conscious applications.)

In this latter paradigm, the unmanaged and managed C++ can seamlessly call each other without cumbersome "glue harnesses" like JNI. The intent is to integrate native C++ with C++ which runs on a vendor-neutral "virtual machine" (i.e. a Virtual Execution System or VES, as it's called in the CLI standard). One challenging aspect of this integration arises on platforms where one or more of the native C++ types are different in size and/or representation from the standardized CLI types; this is a front-burner issue in TG5 today.

Some of the issues considered by TG5 are enhancements that are conceptually simple but make a noticeable difference in ease-of-use. For example, TG5 has requested WG21 to consider changes that would allow two consecutive "close-angle-braces" at the end of template parameter lists such as `list<vector<int>>`, because the C# committee has chosen to implement the new "generics" parameter lists with this ease-of-use enhancement.

Plum Hall is currently developing a test suite for C++/CLI, similar in scope and technique to our existing suites for C, C++, Java, and C#. TG5's current target is completion of the C++/CLI standard by September 2004, about one year from the initial creation of TG5. This is not an imposed requirement, and can be changed by decision of TG5, but reflects our current estimate.

Your TG5 convener welcomes your comments; please contact `tplum@plumhall.com`.

Thomas Plum

[Dedicated to the memory of David Kelly, long-time resident of Oxford, who was still alive when WG21 last met in Oxford.]

Microsoft® Visual C++ .net

Microsoft® Visual Studio® .NET Family Member

This feature was provided by Microsoft as part of their sponsorship of the Spring Conference 2004. ACCU would like to thank Microsoft for their support of this event.

Professionalism in Programming #25

The need for speed (part two)

Pete Goodliffe <pete@cthree.org>

The best is the enemy of the good.

Voltaire

In the first part of this series, we looked at what it means to optimise code, and saw the cases for and against optimisation. In this article, we'll look at the *process* of optimisation. We'll see the correct, methodical approach that will lead to solid, worthwhile code optimisations.

The nuts and bolts

So how do you optimise? Rather than learn a list of specific code optimisations, it's far more important to understand the correct *approach* to optimising. Don't panic; we will see some programming techniques later, but they must be read in the context of this wider optimisation process.

The six steps for speeding up a program are:

1. Determine that it's too slow, and prove you do need to optimise.
2. Identify the slowest code. Target this point.
3. Test the performance of the optimisation target.
4. Optimise the code.
5. Test that the optimised code still works (very important).
6. Test the speed increase, and decide what to do next.

This sounds like quite a performance, but without it you'll waste time and effort, and end up with crippled code that runs no faster. If you're not trying to improve execution speed, adjust this process accordingly; for example tackle memory consumption problems by identifying which data structures are consuming all the memory and target those.

This optimisation procedure is essential; you cannot successfully optimise as you go along. First strive for clear, correct code. Only later look at improving it. But do pay continuous attention to efficiency as you write.

Performance requirements

Certain kinds of system have very strict requirements for program performance; this is especially true of embedded systems controlling critical pieces of hardware.

General purpose software also has performance requirements, although they tend to be specified less rigorously. You won't see many mandated performance requirements – the software is expected to execute in 'reasonable' time on a 'reasonable' computer. The problem comes when you need to know the value of 'reasonable'. A user with a modest PC will have a different opinion than a developer with a cutting edge machine.

The adequate level of performance is usually captured in a product's *Requirements Specification*. This contains a list of all requirements, including those for the program's quality of service. These specifications translate operational requirements into more technical language that describes:

- what is correct behaviour, and
- how we will determine if the software is acceptable.

As well as the expected system behaviour, performance requirements will describe the operational environment (in what conditions the code should work, and when it doesn't matter). Sometimes a *UI Specification* (describing how the user expects the system to respond) indirectly covers a large number of performance requirements.

A good performance specification should be quantitative, not qualitative. This helps us to design acceptance tests based on performance criteria. If a requirement is not verifiable, it is useless.

The requirements specification should not be too prescriptive. If a system is required to position a mechanism accurate to within one millimetre, it is specified. However, the requirements specification does not detail how this criterion is met – it might be a function of the mechanical hardware, of the electronic motor control, or of the controlling software (or more likely, a combination of the three). This is a design decision, often taken much later when the appropriate tradeoffs can be made.

Ensure that you don't write needlessly bloated code. This alone will not lead to code that performs well, but it means that you'll start optimising from a more reasonable place, and will not have to address tedious code issues before getting on to more serious optimisations.

It's important to begin optimisation with a clear goal in sight – the more optimisation you perform, the less readable the code becomes. Know the level of performance you require, and stop when it's sufficiently fast. It's tempting to keep going, trying to continually squeeze out a little extra performance.

To stand any chance of optimising correctly, you must take great care to prevent external factors changing the way your code works. When the world is changing under your feet, you can't compare measurements realistically. There are two essential techniques that help here:

1. Optimise your code separately from any other work, so the outcome of one task doesn't cloud the other.
2. Optimise release builds of your program, not development builds. The development builds may run in a very different manner to release builds, due to the inclusion of debugging trace information, object file symbols, etc.

Now we'll look at each of these optimisation steps in more detail:

Prove you need to optimise

The first thing to do is make sure you really *do* need to optimise. If the code's performance is acceptable then there's no point in tinkering with it. Knuth said (himself quoting C.A.R. Hoare): *"We should forget about small efficiencies, say about 97% of the time: premature optimisation is the root of all evil."* There are so many compelling reasons *not* to optimise that the quickest and safest optimisation technique is to prove that you don't need to do it.

You make this decision based on program requirements or usability studies. With this information you can establish a commercial imperative for optimisation, and determine whether it takes priority over adding new features.

Identify the slowest code

This is the part that most programmers get wrong. If you're going to spend time optimising, you need to target the places where it will make a difference. Investigations show that the average program spends more than 80% of its time in less than 20% of the code [Boehm 87]. This is known as the *80/20 rule*[1]. That's a relatively small target; it's very easy to miss, and spend time optimising code that's rarely run.

You might notice that a part of your program has some relatively easy optimisations, but if it's seldom executed, then there's no point in doing so – clear code is better than faster code in this situation.

So how do you work out where to focus your attention? The most popular – and often the most effective – technique is to use a profiler. This times the flow of control around your program. It shows where that 80% of execution time is going, so you know where to concentrate your effort.

A profiler *doesn't* tell you which parts of the code are slowest; this is a common misconception. It actually tells you where the CPU spends most of its time. This is subtly different[2]. You have to interpret these results, and use your brain. The program might spend most of its execution time in a few perfectly valid functions which cannot be improved at all – it's a sad fact that you can't always optimise. Sometimes the laws of physics win.

There are plenty of benchmarking programs around: many excellent commercial programs, and a number of freely available tools. It's worth spending money on a decent profiler – optimisation can easily eat into your time; this is also an expensive commodity. If you don't have a profiler available, there are a few other timing techniques you can try:

- Put manual timing tests throughout your code. Make sure you use an accurate clock source, and that the time taken to read the clock will not affect program performance too much.
- Count how often each function is called (some debug libraries provide support for this kind of activity).

1 Although some go so far as to claim this should be the 90/10 rule.

2 All code runs at a fixed rate, based on the speed of the CPU clock, the number of other processes being juggled by the OS, and this thread's priority.

- Exploit compiler-supplied 'hooks' to insert your own accounting code when each function is entered/exited. Many compilers provide a means to do this – some profilers are implemented using such a mechanism.
- Sample the program counter; interrupt your program periodically in a debugger to see where control is. This is harder in multithreaded programs, and is a very slow, manual approach. If you have control over the execution environment, you can write scaffolding to automate this kind of test – effectively writing your own form of profiler.
- Test an individual function's impact on the total program execution time by making it slower. If you suspect that a particular function is causing a slowdown, try replacing its call with two calls in succession, and measure how it affects execution time[3]. If the program takes 10% longer to run, then the function consumes approximately 10% of execution time. You can use this as a very basic timing test.

Whilst a profiler (or equivalent) is a good starting point to choose optimisation targets, you can easily miss quite fundamental problems. The profiler only shows how the code in the current design executes – and encourages you to perform code-level improvement only. Look at larger design issues, too. The lack of performance may not be due to a single function, but a more pervasive design flaw. If it is, then you'll have to work harder to remedy the problem. This shows how important it is to get the initial code design right, with knowledge of established performance requirements. Don't rely solely on a profiler to find the causes of program inefficiency – you might miss important problems.

When profiling make sure that you use realistic input data, simulating Real World events. The way your code executes may be drastically affected by the kind of input you feed it, or by the way it is driven, so make sure that you provide true representative input sets. If possible, capture a set of real input data from a live system.

Try profiling several different data sets, to see what difference this makes. Select a very 'basic' set, a 'heavy use' set, and a number of 'general use' sets. This will prevent you optimising for the particular quirks of one input data set. Select profiling test data carefully to represent Real World program use. Otherwise, you might optimise parts of the program that are not normally run.

Using a profiler

Owning a hammer or a saw doesn't instantly make you an expert carpenter; owning a profiler doesn't make you an expert optimiser. You have to know how to use it to perform the task – any tool increases your productivity, it doesn't give you ability.

Although each profiler is different, they follow the same general principles. A profiler is intimately entwined with a particular compiler – it has to reach into the object files and extract the compiler's 'debugging' information to work out which function is currently being run. The basic use of a profiler is:

1. Rebuild your program to support profiling. This usually means linking the executable *without* stripping out debug information. When you do this, you're no longer probing a release build of the code; this contravenes one of the previous optimisation laws. Sigh.
2. Inevitably using a profiler will interfere with running code, but generally this interference will not affect the way the code runs, or skew the timing results too much. We have to live with this.
3. Run the program in the profiler. This is probably an option in your IDE, or a simple profiler command.
4. The profiler will spit out its results when the program finishes. You can browse the profiler's output file, or use a neat graphical tool to interpret this information and display nice graphs.
5. Work out what the results are telling you, and act on them.

The most interesting part of the profiler output is the list of functions that the CPU spent most time in. Use this as a starting point for your optimisation. Read the list, taking into account the number of times each function was called:

- If a function runs very quickly but gets called frequently, try to reduce the number of times it is invoked.
- If a function is only ever called once but takes an age to complete, look for ways to speed up its execution.

Having completed this step, you've found the areas of your code where a performance improvement will have the most benefit. Now it's time to attack them...

Testing the code

We recognised three testing phases in the optimisation procedure. For each piece of code targeted, we test:

- its performance before optimisation,
- that it still works correctly once optimised, and then
- its performance after optimisation.

Programmers often forget the second check: that the optimised code still works correctly in *all* possible situations. It's easy to check the 'normal' mode of operation, but it's not in our nature to test each and every corner case. This can be the cause of weird bugs late in the day, so be very rigorous about this.

You *must* measure the code's performance before and after modification, to make sure that you have made a real difference – and to make sure that it is a change for the better; sometimes an 'optimisation' can be an unwitting *pessimisation*. You can perform these timing tests with your profiler, or by inserting timing instrumentation by hand. Never try to optimise code without performing some kind of before and after measurement.

These are some very important things to think about when running your timing tests:

- Run both the 'before' and 'after' tests with exactly the same set of input data, so that you're testing exactly the same thing. Otherwise your tests are meaningless; you're not comparing like with like. An automated test suite is best[4] – with the same kind of 'live' representative data, we used in the profiling step.
- Run all tests under identical prevailing conditions, so that factors like the CPU load or amount of free memory don't affect your measurements.
- Ensure that your tests don't rely on user input. Humans can cause timings to fluctuate wildly. Automate every possible aspect of the test procedure.

Optimising the code

We'll investigate some specific optimisation techniques in the next article. Speed-ups vary from the simple refactoring of small sections of code to more serious design-level alterations. The trick is to optimise without totally destroying the code.

Determine how many different ways exist to optimise the identified code, and pick the best. Only perform one change at a time; it's less risky, and you'll have a better idea of what improved performance the most. Sometimes it's the least expected things that have the most significant optimisation effects.

After optimisation

We already mentioned performance testing after an optimisation. It's really important to run these before and after performance benchmarks – how else do you know you've made a successful modification?

If an optimisation proves to be unsuccessful, remove it. Back out your changes. This is where a source control system is useful, helping you to revert to the previous code version.

You should also remove the *slightly* successful optimisations. Prefer clear code to modest optimisations (unless you're absolutely desperate for an improvement, and there are no other avenues to explore).

Next time

We'll round off this series by looking at some specific optimisation techniques, and discover how to avoid optimisation altogether.

Pete Goodliffe

References

[Boehm 87] Boehm, Barry, "Improving Software Productivity", 1987, *IEEE Computer*, Vol 20, No 9.

3 This won't necessarily make the function run twice as slowly. File system buffers, or CPU memory caches can enhance the performance of repeated code sections. Treat this as a very rough guide – more qualitative than quantitative.

4 Perhaps you can even use a part of your regression test suite?

A Python Script to Relocate Source Trees

Thomas Guest <thomas.guest@ntlworld.com>

Files form the raw ingredients of a software sytem – source files, build files, configuration files, resource files, scripts etc. These files are organised into directories.

As the system develops, this directory structure must develop with it: maybe an extra level of hierarchy needs adding to accommodate a new revision of an operating system; maybe third party libraries need gathering into a single place; maybe we are porting to a platform which imposes some restriction on file names; or maybe the name originally chosen for a directory has simply become misleading.

This article describes the development of a simple Python script to facilitate relocating a source tree. It is as much – if not more – about getting started with Python as it is about solving the particular problem used as an example.

Statement of the Problem

Let's suppose that we have a directory structure we wish to modify. This existing structure has been reviewed and the decision has been taken to re-map source directories as follows:

```
png         → graphics/thirdparty/png
jpeg        → graphics/thirdparty/jpeg
bitmap      → graphics/common/bitmap
UserIF      → ui
UserIF/Wgts → ui/widgets
os          → platform/os
os/hpux     → platform/os/hpux10
```

By "re-mapped", I mean that the directory and its contents should be recursively moved to the new location. So, for example:

```
UserIF/Wgts/buttons/switchbutton.cpp
    → ui/widgets/buttons/switchbutton.cpp
```

Although it's straightforward create a new top-level directory and copy existing directories to their new locations, the problem we then face is that our source files will no longer build because the files they include have moved. In fact, some of the build files themselves need adjusting, since they too reference moving targets.

An Outline Solution

In outline, our script will implement a two-pass algorithm:

1st Pass: Traverse all files in the current source tree, working out where they will move to. The output of this pass is a container which maps existing files to their new locations.

2nd Pass: For each file found in the first pass, perform the actual relocation, updating any internal references to file paths.

The actual processing for a file in the 2nd pass depends on the type of the file. We can simply copy a bitmap, for example, to its new home, but when relocating a C/C++ source file we'll need to be more careful. This is why I've chosen a two-pass solution: when updating internal file references, I prefer look-up to recalculation.

First Pass – Iterating Over Files

We want to map existing files to their new locations. In Python, the built-in mapping type is called a dictionary. The output of pass one will be a dictionary, which we initialise to be empty.

```
files_map = {}
```

There are two standard modules which support file and directory operations:

`os` – "Miscellaneous operating system interfaces"
`os.path` – "Common pathname manipulations"

Both of these will be of use to our script. In fact, both provide a mechanism for traversing a directory tree:

`os.path.walk`, which calls back a supplied function at each subdirectory, passing that function the subdirectory name and a list of the files it contains.
`os.walk`, which generates a 3-tuple (`dirpath`, `dirnames`, `filenames`) for each subdirectory in the tree.

The second option, `os.walk`, only exists in Python 2.3 (2.3 strengthens the language's support for generators). I prefer it since it makes the script more direct.

```
import os

# Initialise a dictionary to map current file
# path to new file path.
files_map = {}

# Fill the dictionary by remapping all files
# beneath the current working directory.
for (subdir, dirs, files) in os.walk('.'):
  print "Mapping files in subdir [%s]" % subdir
  files_map.update(
           mapFiles(subdir, files)
           )
```

Note the general absence of visible symbols to delimit blocks and expressions – a colon marks the end of the `for` condition, and that's about it. Expressions are terminated by a newline, unless the newline is escaped with a backslash or the expression is waiting for a closing bracket to complete it. Thus the `print` statement terminates at the newline, but the dictionary `update` statement spreads over three lines. Note also that statements can be grouped into a block by placing them at the same indentation level: the body of the for loop is a block of two statements.

To a C/C++ programmer these syntactical rules may seem unusual, dangerous even – attaching meaning to whitespace!? – but I would argue that they actually encourage clean and well laid out scripts.

Incidentally, the default behaviour of the `print` statement is to add a newline after printing. Appending a trailing comma would print a space instead of this newline.

Pass One: Mapping Files and Directories

If we attempt to run the script as it stands, we'll see an exception thrown:

```
<...snip...>
NameError: name 'mapFiles' is not defined
```

which is as we'd expect. We need to define the function:

```
def mapFiles(dirname, files):
  """Return a dictionary mapping files to
                    their new locations."""
  new_dir = mapDirectory(dirname)
  print "mapDirectory [%s] -> [%s]" % \
                         (dirname, new_dir)
  fm = {}
  for f in files:
      fm[os.path.join(dirname, f)] = \
         os.path.join(new_dir, f)
  return fm
```

The Python interpreter needs to know about this function before it can use it, so we'll place it before the path traversal loop.

The first statement of the function body is the function's (optional) documentation string, or docstring. The Python documentation explains why it's worth getting the habit of using docstrings and the conventions for their use.

The function fills a dictionary mapping files to their new location. It uses `os.path.join` from the `os.path` module to construct a file path. The backslash is there to escape a newline, allowing the dictionary item-setter to continue onto a second line.

The final component of the first pass is the function `mapDirectory`, which maps an existing directory to its new location.

The Python Interpreter

If we were to start an interactive Python shell and load the `mapFiles` function, we could then query it and its attributes:

```
>>> mapFiles
<function mapFiles at 0x01106630>
>>> dir(mapFiles)
['__call__', '__class__', '__delattr__',
'__dict__', '__doc__', '__get__',
'__getattribute__', '__hash__', '__init__',
'__module__', '__name__', '__new__',
'__reduce__', '__reduce_ex__', '__repr__',
'__setattr__', '__str__', 'func_closure',
'func_code', 'func_defaults', 'func_dict',
'func_doc', 'func_globals', 'func_name']

>>> mapFiles.func_doc
'Return a dictionary mapping files in dir to
their new locations.'
```

I like to switch between editor and interpreter when developing scripts (on Windows, the PythonWin IDE makes this easy to do, or, alternatively, Python's `-i` option), since it helps me understand both how my script works and how Python works. Here we can see the name "`mapFiles`" refers to a function object which has a list of attributes.

The Python interpreter also allows functions to be exercised immediately:

```
>>> mapFiles('png',('pngRead.h','pngWrite.h'))
```

```
def mapDirectory(dname):
  """Return the new location of the input
                              directory."""
  # The following dictionary maps existing
  # directories to their new locations.
  dirmap = {
    'png'     : 'graphics/thirdparty/png',
    'jpeg'    : 'graphics/thirdparty/jpeg',
    'bitmap'  : 'graphics/common/bitmap',
    'UserIF'  : 'ui',
    'UserIF/Wgts' : 'ui/widgets',
    'os'      : 'platform/os',
    'os/hpux' : 'platform/os/hpux10'
  }
  # Successively reduce the directory path
  # until it matches one of the keys in the
  # dictionary.
  mapped_dir = p = dname
  while p and not p in dirmap:
    p = os.path.dirname(p)

  if p:
    mapped_dir = os.path.join(dirmap[p],
                       dname[len(p) + 1:])

  return mapped_dir
```

The directory rearrangement described earlier in this article has been represented as a dictionary. The input directory is reduced until we match a key in this dictionary. As soon as we find such a match, we construct our return value from the value at this key and the un-matched tail of the input directory; or, if no such match is found, the input value is returned unmodified.

The expression `dname[len(p) + 1:]` is a slice operation applied to a string. Bearing in mind that `p` is the first `len(p)` characters in `dir`, this expression returns what's left of `dname`, omitting the slash which separates the head from the tail of this path.

For example, when mapping the directory '`os/hpux/include`' we would expect to exit the while loop when `p == 'os/hpux'`, and return the result of joining the path `'platform/os/hpux10'` to `'include'`.

```
mapDirectory('os/hpux/include')
   –> os.path.join('platform/os/hpux10',
                   'include')
   –> 'platform/os/hpux10/include'
```

Pass One: Testing

Let's test these expectations by adding the following lines to our script:

```
assert(mapDirectory('os/hpux/include')
     == 'platform/os/hpux10/include')

assert(mapDirectory('os/win32')
     == 'platform/os/win32')

assert(mapDirectory('unittests')
     == 'unittests')
```

These tests pass on unix platforms, but if you run them on Windows the first two tests raise `AssertionError` exceptions (although the final one passes). For now, I'll leave you to work out why – but promise a more platform independent solution in the final version of the script.

Second Pass

Recall that:

2nd Pass: For each file found in the first pass, perform the actual relocation, updating any internal references to file paths.

and that the output of this phase is a dictionary, `files_map`. The main loop for the 2nd pass is:

```
# Create the new root directory for relocated
# files
new_root = '../relocate'
os.makedirs(new_root)

# Now actually perform the relocation
for srcdst in files_map.items():
  relocate(file, new_root, files_map)
```

The function `os.makedirs` recursively creates a directory path. It throws an exception if the directory path already exists. We will not catch this exception since we want to ensure the files are being relocated to a completely new directory.

Pass Two: Relocating by Copying

```
import shutil

def relocate(srcdst, dst_root, files_map):
  """Relocate a file, correcting internal file
                                  references."""
  dst_file = os.path.join(dst_root, srcdst[1])
  dst_dir = os.path.dirname(dst_file)
  if not os.path.isdir(dst_dir):
    os.makedirs(dst_dir)
  if isSourceFile(dst_file):
    relocateSource(srcdst, dst_file,
                   files_map)
  else:
    shutil.copyfile(srcdst[0], dst_file)
```

There aren't many new features to comment on here. The first parameter to the function, `srcdst` is a (key, value) item from the `files_map` dictionary, so `srcdst[0]` is the path to the original file, and `srcdst[1]` is the path to the relocated file, relative to the new root directory.

We create the destination directory unless it already exists. Then, if our file is a source file, we call `relocateSourceFile`; otherwise, we simply copy the file across.

I admit this isn't the most object-oriented of functions. The meaning of the literal `0` and `1` isn't transparent, and switching on type often indicates unfamiliarity with polymorphism. It isn't Python that's to blame here, nor a lack of familiarity with polymorphism: rather a lack of familiarity on my part with Python's support for polymorphism, and a reluctance to add such sophistication to a simple script.

Pass Two: Identifying Source Files

```
import re

def isSourceFile(file):
  """Return True if the input file is a C/C++
                            source file."""
  src_re = re.compile(r'\.(c|h)(pp)?$',
                    re.IGNORECASE)
  return src_re.search(file) is not None
```

We identify source files using a regular expression pattern:

```
r'\.(c|h)(pp)?$'
```

Here, the "r" stands for raw string, which means that backslashes are not handled in any special way by Python – the string literal is passed directly on to the regular expression module.

Regular expression patterns in Python are as powerful, concise and downright confusing to the uninitiated as they are elswhere. I would say that subsequent use of regular expression matches is a little more friendly.

In this case, the regex reads: "match a '`.`' followed by either a '`c`' or an '`h`' followed by one or no '`pp`'s, followed by the end of the string". The `re.IGNORECASE` flag tells the regex compiler to ignore case.

So, we expect:

```
assert(isSourceFile('a.cpp'))
assert(isSourceFile('a.C'))
assert(not isSourceFile('a.cc'))
assert(not isSourceFile('a.cppp'))
assert(isSourceFile('a.cc.h'))
```

This time, the assertions hold.

Pass Two: Relocating Source Files

```
def relocateSource(srcdst, dstfile,
                   files_map):
  """Relocate a source file, correcting
        included file paths to included."""
  fin  = file(srcdst[0], 'r')
  fout = file(dstfile, 'w')
  for line in fin
    fout.write(processSourceLine(line,
                      srcdst, files_map))
  fin.close()
  fout.close()
```

The function `relocateSource()` simply reads the input file line by line. Each line is converted and written to the output file.

Pass Two: Processing a Line of a Source File

```
def processSourceLine(line, srcdst,
                      files_map):
  """Process a line from a source file,
         correcting included file paths."""
  include_re = re.compile(
        r'^\s*#\s*include\s*"'
        r'(?P<inc_file>\S+)'
        '"')
  match = include_re.match(line)
  if match:
    mapped_inc_file = mapIncludeFile(
        match.group('inc_file'),
        srcdst,
        files_map)
    line = line[:match.start('inc_file')] + \
           mapped_inc_file + \
           line[match.end('inc_file'):]

  return line
```

The function `processSourceLine` has a rather more complicated regex at its core. Essentially we want to spot lines similar to:

```
#include "UserIF/Wgts/Menu.hpp"
```

and extract the double-quoted file path. The complication is that there may be any amount of whitespace at several points on the line – hence the appearances of `\s*`, which reads "zero or more whitespace characters".

The three raw strings which comprise the regex will be concatenated before the regex is compiled - in the same way that adjacent string literals in C/C++ get joined together in an early phase of compilation. I have split the string in this way to emphasise its meaning.

The bizarre `(?P<inc_file>\S+)` syntax creates a named group: essentially, it allows us to identify the sub-group of a match object using "`inc_file`".

So, the function looks for lines of the form:

```
#include "inc_file"
```

then calls `mapIncludeFile(inc_file...)` to find what should now be included, and returns:

```
#include "mapped_inc_file"
```

Incidentally, I am assuming here that the angle brackets are reserved for inclusion of standard library files – or at least not the files we are moving. That is, we don't try and alter lines such as:

```
#include <vector>
```

Pass Two: Mapping Include Files

```
import sys

def mapIncludeFile(inc, srcdst, files_map):
  """Determine the remapped include file
                                   path."""
  # First, obtain a path to the include file
  # relative to the original source root
  if os.path.dirname(inc):
    pass # Assumption 1) - "inc" is our
         # relative path
  else:
    # Assumption 2) The file must be located
    # in the same directory as the source file
    # which includes it.
    inc = os.path.join(
              os.path.dirname(srcdst[0]), inc)
  # Look up the new home for the file
  try:
    mapped_inc = files_map[inc]
    if (os.path.dirname(mapped_inc) ==
        os.path.dirname(srcdst[1])):
      mapped_inc = os.path.basename(mapped_inc)
  except KeyError:
    print 'Failed to locate [%s] (included ' \
          'by [%s]) ' \
          'relative to source root.' % (
          include, srcdst[0])
    sys.exit(1)
  return mapped_inc
```

The function `mapIncludeFile` is actually quite simple, though only because of an assumption I have made about the way include paths are used in this source tree. The assumption is:

> All `#include` directives give a path name relative to the root of the source tree, except when the included file is present in the same directory as the source file – in which case the file can be included directly by its basename. Furthermore, there are no source files at the top-level of the source tree (there are only directories at this level).

For source trees with more complex include paths, and correspondingly more subtle `#include` directives, this function will need fairly heavy-weight adaptation. (Alternatively, run another script to simplify your include paths first.)

If this assumption holds, we can easily determine the original path to the included files, then use our files_map dictionary to look up the new path. If the assumption doesn't hold, then the dictionary look up will fail, raising a `KeyError` exception. The exception is caught, a diagnostic printed, then the script exits with status `1`.

We could test whether "`mapped_inc`" is a key in our dictionary before attempting to use it; and if it were absent, we could simply print an error and continue. However, we choose to view such an absence as exceptional since it undermines the assumptions made by the script. We do not wish to risk moving thousands of files without being sure of what we're doing.

Finishing Touches

The final script appears at the end of this article.

The Windows problem I mentioned earlier is caused by the platform specific directory path separator. Unix uses forward slashes, Windows backslashes. My chosen solution is to work with "normalised" paths internally until we actually write out the include files, when we make sure forward slashes are used as separators.

As a gesture towards user-friendliness I have added an indication of progress and an output log. However, this script remains very much for software developers who understand what it's doing. I have chosen "`sys.stdout.write`" in preference to the "`print`" statement used during the script's development, since it gives greater control over output format.

I have not done anything special with Makefiles, project files, Jamfiles – or whatever else you use with your build system. There will be an order of magnitude fewer of these to deal with (unless you have a very strange build system), but the same techniques apply.

The script as it stands has several weaknesses. It is, of course, suited to doing a very specific job: solving the exact problem laid out at the start of this article, right down to the specified directory mapping. Whilst it would be overkill to provide a GUI allowing users to enter this mapping, this input data could usefully be separated from the body of the script. Similarly, as already mentioned, the script makes some big assumptions about way include paths are used in this particular system. Finally, there are no unit tests – the only testing has been a rather ad hoc probing of functions during the script's development.

Concluding Thoughts

This sort of bulk re-arrangement of files is well suited to a scripted solution for reasons of:

Reliability: The script can be shown to work by unit tests and by system tests on small data sets. Then it can be left to do its job.

Efficiency: Editing dozens – perhaps hundreds – of files by hand is error prone and tedious. What's worse, unless some moratorium on check-ins has been imposed during the restructure, the new structure may be out of date before it is ready. A script can process megabytes of source in minutes. Alternatively, it will happy to run at night after even the most nocturnal of programmers has logged out.

Recordability: The script becomes part of the source tree (perhaps in the tools/scripts directory), in which place it can record accurately and repeatably the tasks it performs.

Reusability: A well-written script can be amended and enhanced to solve future source re-organisations. And even if it can't be re-used, a knowledge of scripting can.

Thomas Guest

References

[1] `http://www.python.org` – the official website for the Python language.

Credits

[1] I used the PythonWin IDE (available with the ActivePython distribution) while developing this script, and can recommend it to Windows users. The context sensitive help is great, the debugger is good, and the key-bindings – incredibly – include my favourites from both GNU emacs and Microsoft Visual Studio.
ActivePython: `http://www.activestate.com/Products/ActivePython/`

[2] Thanks to Dan Tallis for reviewing an earlier draft of this article, and for inspiring me to learn Python.

```
import os
import re
import shutil
import sys

def mapDirectory(dirmap, dname):
  """Return new location of the input directory."""
  # Successively reduce the directory path until it
  # matches one of the keys in the directory map.
  mapped_dir = p = dname
  while p and not p in dirmap:
    p = os.path.dirname(p)

  if p:
    mapped_dir = os.path.join(dirmap[p],
                              dname[len(p) + 1:])
  return mapped_dir

def mapFiles(logfp, dirmap, dname, files):
  """Return a dictionary mapping files in dir to
                              their new locations."""
  dname = os.path.normpath(dname)
  new_dir = mapDirectory(dirmap, dname)
  logfp.write("mapDirectory [%s] -> [%s]\n" % \
          (dname, new_dir))
  fm = {}
  for f in files:
    src = os.path.join(dname, f)
    dst = os.path.join(new_dir, f)
    logfp.write("\t[%s] -> [%s]\n" % (src, dst))
    fm[src] = dst
  return fm
```

```
def isSourceFile(file):
  """Return True if the input file is a C/C++ source
                                              file."""
  src_re = re.compile(r'\.(c|h)(pp)?$',
                      re.IGNORECASE)
  return src_re.search(file) is not None

def swapSlashes(str):
  """Return the input string with backslashes swapped
                                  to forward slashes."""
  back_to_fwd_re = re.compile(r'\\')
  return back_to_fwd_re.sub('/', str)

def mapIncludeFile(logfp, inc, srcdst, files_map):
  """Determine the remapped include file path."""
  # First, obtain a path to the include file
  # relative to the original source root
  if os.path.dirname(inc):
    pass  # Assumption 1) - "inc" is our relative
          # path
  else:
    # Assumption 2) The file must be located in the
    # same directory as the source file which
    # includes it.
    inc = os.path.join(os.path.dirname(srcdst[0]),
                       inc)
    inc = os.path.normpath(inc)
  # Look up the new home for the file
  try:
    mapped_inc = files_map[inc]
  except KeyError:
    err_msg= ('\nFatal error: Failed to locate [%s] '
        '(included by [%s]) '
        'relative to source root.' %
        (inc, srcdst[0]))
    logfp.write(err_msg)
    sys.stderr.write(err_msg)
    sys.exit(1)

  if (os.path.dirname(mapped_inc) ==
      os.path.dirname(srcdst[1])):
    mapped_inc = os.path.basename(mapped_inc)

  return mapped_inc

def processSourceLine(logfp, line, srcdst,
                      files_map):
  """Process a line from a source file,
                correcting included file paths."""
  include_re = re.compile(
        r'^\s*#\s*include\s*"'
        r'(?P<inc_file>\S+)'
        '"')
  match = include_re.match(line)
  if match:
    logfp.write('    [%s] -> ' % line.rstrip())
    mapped_inc_file = mapIncludeFile(
         logfp,
         match.group('inc_file'),
         srcdst,
         files_map
         )
    line = line[:match.start('inc_file')] + \
               mapped_inc_file + \
               line[match.end('inc_file'):]
    line = swapSlashes(line)
    logfp.write('[%s]\n' % line.rstrip())

  return line

def relocateSource(logfp, srcdst, dstfile,
                   files_map):
  """Relocate a source file, correcting paths
                              to included files."""
  infp  = open(srcdst[0], 'r')
  outfp = open(dstfile, 'w')
  logfp.write('Relocating source file [%s] -> [%s]\n' %
            (srcdst[0], dstfile))
  for line in infp:
    outfp.write(
          processSourceLine(
              logfp, line, srcdst, files_map)
          )
  infp.close()
  outfp.close()

def relocate(logfp, srcdst, dst_root, files_map):
  """Relocate a file, correcting internal file
                                     references."""
  dst_file = os.path.join(dst_root, srcdst[1])
  dst_dir = os.path.dirname(dst_file)
  if not os.path.isdir(dst_dir):
    os.makedirs(dst_dir)
  if isSourceFile(dst_file):
    relocateSource(logfp, srcdst,
                   dst_file, files_map)
  else:
    logfp.write('Copying [%s] -> [%s]\n' %
              (srcdst[0], dst_file))
    shutil.copyfile(srcdst[0], dst_file)

def percent(num, denom):
  """Return num / denom expressed as an integer
                                     percentage."""
  return int(num * 100 / denom)
```

```
def printSettings(fp, dmap, src, dst):
  """Output script settings to the input file."""
  fp.write('Relocating source tree from [%s] '
           'to [%s]\n' %
           (src, dst))
  fp.write('Relocating directories:\n')
  for d in dirmap:
    fp.write('    [%s] -> [%s]\n' %
             (d, dmap[d]))
  fp.write('\n')


# Main processing starts here...

# First, set up script data:
#   - a dictionary mapping existing dirs to
#       their new locations,
#   - the source and destination roots for
#       the source tree,
np = os.path.normpath
dirmap = {
  np('png')     : np('graphics/thirdparty/png'),
  np('jpeg')    : np('graphics/thirdparty/jpeg'),
  np('bitmap')  : np('graphics/common/bitmap'),
  np('UserIF')  : np('ui'),
  np('UserIF/Wgts') : np('ui/widgets'),
  np('os')      : np('platform/os'),
  np('os/hpux') : np('platform/os/hpux10')
}

from_root = '.'
to_root = '../relocate'

# Further initialisation.
logfp = open('relocate.log', 'w')
printSettings(logfp, dirmap, from_root, to_root)
printSettings(sys.stdout, dirmap, from_root, to_root)

# Initialise a dictionary to map current file path
# to new file path.
files_map = {}

# Fill the dictionary by remapping all files beneath
# the current working directory.
sys.stdout.write('Preprocessing files. '
                 'Please wait.\n')

for (subdir, dirs, files) in os.walk(from_root):
  files_map.update(
      mapFiles(logfp, dirmap, subdir, files)
      )

# Create the new root directory for relocated files
os.makedirs(to_root)

# Now actually perform the relocation
count = len(files_map)
item = 0
sys.stdout.write('Relocating [%d] files.\n'
                 'Logfile at [%s]\n'
                 'Progress [%02d%%]' %
      (count, logfp.name, percent(item, count))
    )

for srcdst in files_map.items():
  item += 1
  sys.stdout.write('\b\b\b\b%02d%%]' %
                   percent(item, count))
  relocate(logfp, srcdst, to_root, files_map)

logfp.close()
sys.stdout.write('\nRelocation completed '
                 'successfully.\n')
```

I_mean_something_to_ somebody, Part Two

Derek Jones <derek@knosof.co.uk>

Introduction

This is the second of a two part article describing an experiment carried out during the 2003 ACCU conference. The first part was published in a previous issue of C Vu (15.6, December 2003) and discussed the background to the experiment and some of the applicable characteristics of the subjects taking part; this one, the second, discusses the results of the experiment.

The aim of this experiment was to measure one particular aspect of software developers' behaviour when assigning meaning to identifier names. This aspect was the extent to which knowledge of the application domain of the source code containing an identifier affects the meaning developers assign to that identifier name.

Software developers are constantly exhorted to use *'meaningful'* identifier names. However, there have not been any published studies investigating the kinds of information readers extract from identifier names or of any benefits the availability of this information might provide to readers. Reading source code whose identifier names are based on a human language the reader does not speak provides a vivid example of the often unappreciated benefit that identifier names can provide to readers (when these names are based on a human language spoken by the reader).

```
  if(pParametreFichier != (FILE*)NULL) {
    memset(&Enregistrement.CodeInterne1, '\0',
           sizeof(Enregistrement.CodeInterne1));
    memset(&Enregistrement.BlocPrimaireNumerique ,
           '\0', sizeof(Enregistrement.
                              BlocPrimaireNumerique));
    while(!ExcTrouve)
      ...
```

Background

Words are used both to communicate with other people and for internal thought processes. The culture we are born into provides us with a predefined set of words and a network of meanings associated with them. The use of words in their spoken form to communicate with other people has a cost that speakers attempt to minimise by using them in a way that is consistent with the meaning they believe their listeners will assign to them. A lifetime of realtime feedback from the people spoken to enables users of a language to build a detailed collection of beliefs on the meanings assigned to words by both people in general and some specialist groups of people (e.g., software engineers).

When speaking it is expected that not only will listeners make an effort to comprehend the speakers' thought processes, but that speakers will make an effort to ensure that what they are saying is comprehensible to their listeners. When writing text people must make use of their experience with the spoken form to help ensure that readers will assign a meaning to the words that is consistent with that intended. However, there is no realtime feedback between writer and reader[1] and experience shows that readers often have to invest significantly more effort to assign a coherent meaning to what they read, compared to the effort needed while listening during a spoken conversation.

Software developers are not usually told which identifiers they should use in a given context and are rarely given rules for creating new identifier names from existing ones.[2]

Selecting identifiers

Experience shows that many developers believe that the names they select for identifiers are *'obvious'*, *'self-evident'*, or *'natural'*. Studies of people's performance in creating names for objects suggests that this belief is false [2, 4, 5]. When asked to provide names for various kinds of entities people have been found to select a wide variety of different names, showing that there is nothing *'obvious'* about the choice of a name.

One naming study [4, 5] described operations (e.g., hypothetical text editing commands, categories in *'Swap 'n Sale'* classified ads, keywords for recipes) to subjects, who were not domain experts, and asked them to suggest a name for each operation. The results showed that the name selected by one subject was, on average, different from the name selected by 80-90% of the other subjects (one experiment included subjects who were domain experts and the results for those subjects were also consistent with this performance). The number of occurrences of different names chosen tended to follow an inverse law with a few words occurring frequently and most only rarely.

Various factors have been found to influence the selection of what is believed to be the appropriate word in a given context. A study by Labov [7] showed subjects pictures of individual items that could be classified as either cups or bowls, as shown in Figure 1. These items were presented in one of two contexts; a neutral context in which the pictures were simply presented and a food context (subjects were asked to think of the items as being filled with mashed potatoes).

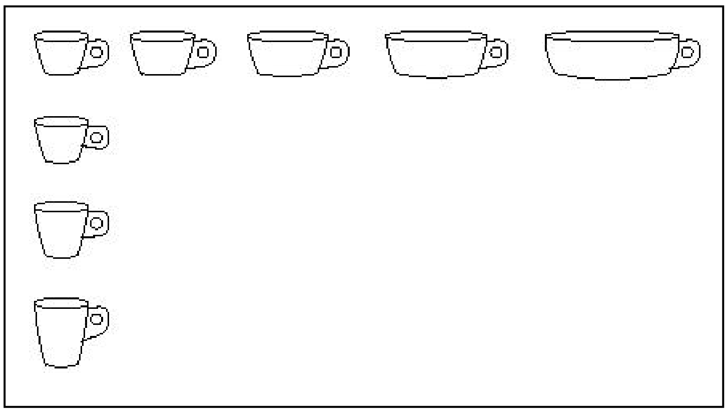


Figure 1: Cup and bowl like objects of various widths (ratios 1.2, 1.5, 1.9, and 2.5) and heights (ratios 1.2, 1.5, 1.9, and 2.4). From Labov [7].

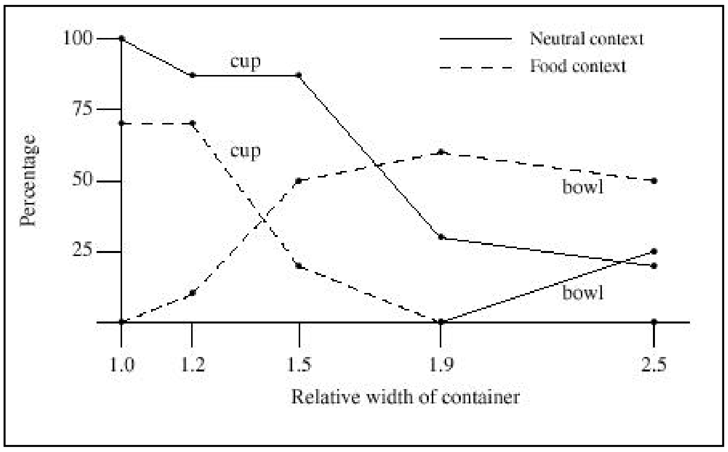


Figure 2: The percentage of subjects who selected the term 'cup' or 'bowl' to describe the object they were shown (the paper did not explain why the figures do not sum to 100%). From Labov [7].

The results (Figure 2) showed that as the width of the item seen was increased, an increasing number of subjects classified it as a bowl. By introducing a food context subject responses are shifted towards classifying the item as a bowl at narrower widths.

Recognizing words

Human languages have a relatively fixed set of letter sequences that are acknowledge by speakers of a language as being *'root words'* (this glosses over the heated discussions that sometimes occur over what letter sequences should be treated as root words). Additional words can be derived from these words using language specific rules (e.g., *write* → *writes*, *writing*, *written*; *writer* could be treated as either a derived or a root word).

Identifiers sometimes contain more than one word. In this case readers need to either use their knowledge of existing words to subdivide an identifier's character sequences, or use deduction based on common naming conventions to extract words (e.g., `IsHot` is likely to be interpreted as the phrase 'is hot', rather than 'I shot').

Identifiers often have the form of one or more abbreviated words. A study by Ehrenreich and Porcu [3] found that readers' performance in

1 'Talking' via text messaging is not discussed here.

2 The high cost of having database fields representing the same data item, e.g., a person's first name, but with different names, e.g., first_name or given_name or christian_name, across multiple databases has caused some organizations to plan to start mandating the use of specific names to denote specific data items [1].

reconstructing the original word, from an abbreviated form, was significantly better when they knew the rules used to create the abbreviation (81-92% correct), compared to when the abbreviation rules were not known (at best 62% after six exposures to the letter sequences). Given that this experiment was not intended to measure subjects' abbreviation to word reconstruction performance, no rarely occurring abbreviations were used.

Studies of meaning assignment

While there have been no other published studies of how people assign a meaning to identifiers there have been a few studies of a similar nature for words.

A study by Nickerson and Cartwright [9] asked subjects to write down as many different meanings of a word (presented one at a time, in written form, for 30 seconds). Combining the results from all subjects showed that words were often given over 6 and sometimes as many as 20 different meanings. The majority of the responses for a given word were usually contained within one or two meanings.

Word association is an activity that has some similarities to providing a meaning for a word. Studies of word association give subjects a word and ask them to write down the first meaningfully related word that comes to mind. (e.g., *doctor* ? *nurse*).

The results of these studies[3] have found that there is rarely a single answer, a wide range of responses is given, and words given by subjects do not always overlap those of other subjects

A subject's age has also been found to be a factor in word association performance. A study by Hirsh and Tree [6] compared the responses of young (21-30) and older (66-81) adults to 90 stimulus words. The results showed that the same word was produced as the most popular response, for a given age group, in 36 out of 90 cases (when the top three responses were considered the overlap between groups was 57%). They also found that the younger group produced a wider range of responses, and that members of the older group were much more likely to select the most popular response for their group (40%, against 20% for the younger group).

Experimental setup

The experiment was performed during two 30 minute sessions on different days of the 2003 ACCU conference held in Oxford, UK. Subjects were given a brief introduction to the experiment, during which they filled out background information about themselves. They then spent 15 minutes working on the identifier list. All subjects volunteered their time and were anonymous.

The first part of this paper describes the background of the subjects and how this information was collected.

Almost any sequence of characters could serve as an identifier. However, the initial list of identifiers considered for use in the experiment were obtained by extracting all identifiers that were common to the source code of a variety of programs. These programs were the Linux kernel, the game Doom, gcc (the GNU compiler collection), Netscape internet browser, PostgresSQL database, AFS (Advanced File System) from IBM, and OpenMotif from the OpenGroup. It was hoped that usage in a wide variety of programs was an indication that an identifier had a significant meaning to a large number of developers. This method also removes experimenter bias from the choice of identifier names (but not from the choice of programs to consider).

The initial list was refined by removing those identifiers that were the names of standard library functions (these might be recognized as such and their library meaning given as a response), or contained rarely occurring abbreviations, or contained a single character. The resulting list of identifiers was randomized and printed one per line on A4 sheets of paper.

All subjects from both groups saw an identical list of identifiers. However, one group was told that the identifiers came from a multiplayer game, while the other that they came from the Linux kernel. The instructions given were:

> The following pages contain identifiers that have been extracted from the source of {a very large multiplayer game program}/{the Linux kernel}. For each identifier:
> 1 when you first see the identifier, write down any ideas that pop into your head about what it might represent,
> 2 briefly (5-10 seconds is sufficient) think about what the identifier might represent. Write any new ideas you have on a separate line.

3 The University of South Florida word association norms [8] lists nearly three-quarters of a million responses to 5,019 stimulus words produced by 6,000 participants.

Threats to validity

There are a number of reasons why the responses given in this experiment might not be valid in a source code comprehension context. These include:

- developers are not usually asked to provide the kind of information that they were asked to provide in this experiment. It is possible that the subjects were unsure of the responses expected of them, or misinterpreted the instructions they were given,
- identifiers invariably exist within a context when they are read in source code. For instance, there are other identifiers (e.g., the name of the function in which an identifier is referenced) whose names often provide a subcontext,
- providing a possible meaning for an identifier requires a lot of intellectual effort. It is unusual for developers to be asked to provide a meaning to so many identifiers over such a relatively short period of time. Over the period of the experiment fatigue may have caused subjects' performance to decline, because of the high cognitive work load.

A few of the subjects had a different cultural background from the majority of the subjects (i.e., they were not British). It is possible that these subjects made use of different cultural conventions when assigning meaning to identifiers. For instance, in the US politicians *run* for office, while in Spain and France they *walk*, and in Britain they *stand* for office.

It is possible that on the first day I failed to point out during the introduction that the identifiers were extracted from a multiplayer game (I did point out that the identifiers came from the Linux kernel on the second day). This information is given in the instructions, but it is possible that subjects did not read the sentence containing this information.

Results

The 45 subjects produced a total of 1662 responses (34.8% Linux, 65.2% game), and 74 different words were responded to. There were 179 responses (45 different words) where the subject had written "none" (or a question mark, or a dash). The identifiers were printed on both sides of the page and some subjects only gave responses for identifiers appearing on the odd numbered pages. In this case the identifiers appearing on the even numbered pages were not counted as "none". See Table 1 on the next page for a summary of responses.

Each subject's response for each identifier needed to be classified. The following process was intended to ensure that the person doing the classification (your author) was not influenced by information about the subject who gave the response. (i.e., whether the subject belonged to the Linux or games group, and which responses were given by the same subject). Every response was automatically assigned a random number and the resulting list of identifier/response pairs was sorted. This list of randomised responses was the one used for classification.

Certain words and phrases occurred several times in the responses and were assumed to imply a game context, but not a Linux context. These included: *player*, *game*, *skill level*, and *shoot*. While some words appear to have an obvious games meaning (i.e., *kill*), if it was possible that they also had a Linux meaning they were not classified as being games related.

Words and phrases that might be claimed to be a strong indication of a Linux context (e.g.,*Linux*, *operating system*) rarely occurred in the responses. Much of the functionality provided by an operating system (e.g., Linux) might reasonably also be expected to be provided internally within a game. For instance, *virtual memory* refers to a memory management mechanism used by both operating systems and games (which, for efficiency reasons, might swap unneeded game information out of fast memory). This overlap in functionality, which many subjects are likely to be aware of, makes it difficult to reliably classify any responses as belonging to a Linux context.

A games context was assigned to 134 responses (12.4% of responses made by games subjects) scattered over 33 different words. A Linux context was assigned to 10 responses (1.2% of responses made by Linux subjects) scattered over 6 different words.

The forms of the meanings given were such that it was rarely possible to definitely specify which group a response belonged to. For instance, for the identifier `blue_pos` many subjects gave a response of the form *position of some blue thing*. In itself this response is not sufficient to be able to assign a Linux or game context. Additional information such as *index into array* could apply in either context, while use of the word*player* would suggest a game context.

In many cases the responses described a possible role that the identifier might fill, e.g., flag, or counter, while in other cases subjects simply expanded an identifier to a non-abbreviated form, e.g., gave *page number* as the response to `pagenum`.

The responses contained fewer different meanings per identifier than the Nickerson and Cartwright [9] study. However, this experiment did not explicitly request subjects to list all possible meanings of an identifier.

Discussion

This study set out to investigate the extent to which knowledge of the applicable application domain affected the meaning assigned to identifier names. A single experiment was performed, resulting in a single data point. More measurements, based on responses for other identifiers and application domains, are needed before it is possible to draw any general conclusions about the interaction between developer knowledge of the application domain and the meaning assigned to identifiers.

However, the 12.4% of game subject responses having a game context is significantly less than 100%. Some of the possible reasons for this include:

- subjects implicitly knew that many identifiers appearing in source code have no direct connection to the application domain. That is to say, many identifiers are used in the implementation of some algorithm and the choice of their names is primarily influenced by this algorithmic context. The meanings assigned to identifiers reflected this developer knowledge of typical identifier usage patterns,
- a failure by subjects to provide all of the information needed by this study. It is possible that the large number of identifiers appearing in the handout and the short amount of time available led to subjects deciding to provide brief, rather than detailed, responses. Subjects were not aware of the exact nature of the experiment or the kind of information it was hoped they would provide.

A "flag" meaning was given in a surprising number of responses. This may represent a default response, given when subjects could not think of anything else to write, or perhaps the identifier names used in this experiment often have this meaning in source code.

The responses generally involved concepts encountered in software engineering.

Conclusion

As the first of its kind the results of this experiment encountered a number of problems:

- feedback from subjects suggested that in the short space of time available they were not able to reliably estimate the quantity of code read/written. Given that few developers regularly measure the amount of source they have read/written it is not clear that anybody would be able to provide a reasonably accurate answer to this question,
- many of the written responses provided by subjects had a low information content (i.e., the question being asked was not answered). Providing subjects with more time and asking them to provide a detailed response, or interviewing subjects on a one-to-one basis would solve this problem,
- feedback from subjects suggested that without the context of the surrounding code it was difficult to provide what they considered to be a good interpretation of the likely meaning of an identifier's name,
- choosing identifiers based on their occurrence in various programs may prevent experimenter bias and provide a good justification for their use, but it severely restricts the semantic range of identifiers that can be used.

Derek Jones

Further reading

In many ways identifiers are metaphors. For a fascinating introduction to metaphors in English see: "Metaphors we live by" by G. Lakoff and M. Johnson.

For an interesting, and readable, discussion of people's performance in answering questions they do not know the answer to see: "Simple heuristics that make us smart" by G. Gigerenzer and P. M. Todd.

[concluded at foot of next page]

Identifier	Number of Responses	Responses (number)
accurate	27	flag (12), none (6), numeric value (4), game (3)
answer	29	input value (9), result (4), none (4), string value (2), game (1)
blue_pos	42	game (15), none (14), position of (10)
body	30	none (7), game (7), code (7), html (3)
children	19	tree structure (6), processes (3), OO (3), counter (2), none (1)
cur_mode	40	cursor (4), current mode (24), none (2), game (2), linux (1)
def	19	definition (6), define (6), none (2), language preprocessor (2)
digest	44	cryptography (12), summary (9), eat (8), none (6), game (3)
disconnected	28	flag (most), not connected (1), game (1)
driver	38	device driver (15), game (5), none (4)
drop	44	delete/discard (11), game (8), none (4), connection (4)
event_mask	21	bit map/mask (all)
force	22	physical force (8), none (4), flag (4)
fraction	23	mathematical (16), ration (2), none (2)
fragstotal	32	total fragments (12), game (10), memory fragments (2), 'frags' (2), none (1)
inactive	24	flag (most), none (1), game (1)
inc	33	increment (most), none (3), include (3)
last_sent	40	time message sent (most), none (2)
levels	32	level count (most), game (8), none (3)
Lock	44	concurrency (most), game (2), none (1), lake (1)
magnitude	45	size of (most), absolute value (5), none (3), game (3)
mirror	45	copy/cache/backup (most), none (6), game (5)
misses	34	count of (most), game (6), cache (4), wife (1), none (1)
near	39	close (11), shortptr (8), none (8), game (2)
numsegs	44	number of segments (most), linux (4), game (2), none (1)
origin	24	coordinates (most), parent (1)
outside	39	none (9), flag (7), game (4), linux (1)
pagenum	45	number (15), document (14), memory (3), counter (3), none (2)
picture	24	image (13), pointer (3), none (2), Cobol (1)
play	45	sound (14), start something (11), game (8), none (2)
position	35	location/coordinates (most), in list (5), game (5)
purge	41	clean out (13), delete (12)
quick	44	flag (most), none (10), fast (9), game (4), optimization (3)
registered	45	flag (most), registration (6), signed on (4), Linux (2), game (1)
reliable	43	none (9), trustworthy (5), correct (4), communication link (3), game (2)
routine	22	function (9), none (6), ordinary normal (3)
rover	32	none (14), dog (6), data structure (3), car (3)
self	44	this (18), object (18), game (6)
single	35	none (8), game (6), singleton (5), flag (3)
stopped	44	process (11), finished (6), none (2), game (2), flag (1)
transformed	43	none (2), game (2), flag (1), changed (1)
translation	40	language (12), cartesian (6), none (4)

Table 1: Responses. The five most common responses for identifiers having more than 20 responses ("most" indicates that most responses had this form).

Writing Maintainable Code

Anthony Williams <anthony_w@onetel.net.uk>

Recently, I've been thinking hard about what makes code maintainable, and how to write code to be maintainable. This interest has partly been driven by the mentoring of those starting out in C++ that I've been doing, both through the ACCU mentored developers program, and for work.

The principles I've identified have not really been hidden; since they've been widely documented for years, and they're actually things that most good developers do as a matter of course. However, as with many things, you don't necessarily realize their benefits until you rediscover them yourself.

Long Functions

Some of the things that have shown themselves to be useful are relatively obvious when you think about it, but very easy to not do. For example, long functions can easily make code very hard to understand, especially if they end up many times the number of lines visible on one page in your editor. In fact, at modern screen resolutions a function that fills one page is probably too long (there are 70 lines visible at once in my editor at the screen resolution I use, for example). However, it is very easy to write long functions, especially when maintaining old code – you need to add some extra processing *here*, but you don't want to disturb the structure of the code too much, so you just write it as part of the same function. Alternatively, you start off with a `switch` statement that has a few `case` labels, each of which does something small, and over time it grows into a huge monolithic monster with lots of `case` labels, each of which has a page of code attached.

Long functions often arise when a function has too much responsibility – it is trying to do more than one thing. If this is the case, then it will probably be relatively easy to split it into several smaller functions: do *this*, then *that*, and possibly *that* as well. Taking the monolithic `switch` statement as an example, it would probably be simpler to understand if each `case` just invoked an appropriate `doXXX` function which contained the attached page of code. It might even be improved by replacing it with a table-driven approach.

Multiple Responsibilities

It is not just functions that can have too much responsibility; the same applies to classes and modules as well. It is a common scenario for the same class to be responsible for processing data, displaying it to the user, and handling user input events, but such a mix can easily lead to spaghetti code where it is hard to see where the boundaries lie. This is what is referred to when people talk about the *cohesion* of the code – code with high cohesion has a clearly defined responsibility, and it is clear how it manages that responsibility. Having multiple responsibilities can also affect exception safety – it is very hard to write exception-safe code that manages two resources, unless you divide up the responsibility and have two objects (maybe member sub-objects of the original), each of which manages one resource. This is the idea behind the adage "Do one thing well", the side effect of which is that the code becomes more reusable – there is more chance that you will need code that does precisely the same thing elsewhere (whether in the same application, or in another application) if the thing that it does is small and well-defined.

Just as an example, I recently found that managing Windows Combo Box controls was so much easier once I had written a few small functions to populate the boxes and retrieve their data. All these functions did was wrap the two or three API calls required in each case, but they greatly simplified the code that uses them, and they are used *everywhere* in my applications, wherever Combo Box controls are used. (Aside: the main application where I made this change was using MFC; given that MFC is such a *huge* framework, it surprises me that these things are still so complicated)

Don't Repeat Yourself

This example also illustrates another principle – duplicate code is bad. Often quoted as "Don't repeat yourself" (DRY), or "Once and only once" (OAOO), the idea here is that having the same code in more than one place is just asking for problems – if you make a mistake, and the code needs fixing, then you have to remember to fix it everywhere. Assuming you duplicated it correctly in the first place, that is. If instead you create a class or function that does what you want, and use it everywhere it is needed, then the benefits are twofold – not only is there only one place to fix the code if there are any problems, but you've increased the expressiveness of the places that use it (assuming you've chosen a sensible name for your class or function). Rather than each occurrence of the code being complicated by the actual technical details of the duplicate code, there is instead a function call or a class object that documents the intent of what is to be achieved. Oh, and it probably makes the client code have shorter functions, too.

A consequence of removing duplication ruthlessly is that you end up with less code overall, and that that is there is clearer, so the whole code-base is easier to understand.

Many refactorings (see later) assist with duplication removal – by Extracting a Method you can then call it from other places that do similar things, for example.

A Rose by any other name...

Once you've broken down your classes and functions into lots of small classes and functions, that Do One Thing Well, it is important to give them good names. Names are a form of documentation that is part of the code. They are better than comments, but they can be just as helpful in aiding the understanding of code as a well written comment or API document. Good naming might include providing named constants for intermediate steps in a complex expression, rather than just relying on temporaries.

A rose by any other name might smell as sweet, but you don't want to have to smell it to find out.

[continued from previous page]

The University of South Florida word association norms can be downloaded from: `http://w3.usf.edu/FreeAssociation`

The responses given in the experiment (stripped of subject background information) can be downloaded from: `http://www.knosof.co.uk/cbook/accu2003.html`

Acknowledgements

The author wishes to thank everybody who volunteered their time to take part in the experiment and the ACCU conference organizers for making conference slots available to run it.

References

[1] Justice Standards Clearinghouse: `http://it.ojp.gov/jsr/public/index.jsp`, 2004

[2] J.M.Carroll, *What's in a Name? An essay on the psychology of reference*, W.H.Freeman, 1985

[3] S.L.Ehrenreich and T.Porcu, "Abbreviations for automated systems: Teaching operators the rules", in A.Badre and B.Shneiderman, editors, *Directions in Human/Computer Interaction*, chapter 6, pages 111-135, Ablex Publishing Corp., 1982

[4] G.W.Furnas, T.K.Landauer, L.M.Gomez, and S.T.Dumais, "Statistical semantics: Analysis of the potential performance of key-word information systems", *The Bell System Technical Journal*, 62(6):1753-1805, 1983

[5] G.W.Furnas, T.K.Landauer, L.M.Gomez, and S.T.Dumais, "The vocabulary problem in human-system communication: an analysis and a solution", *Communications of the ACM*, 30(11):964-971, 1987

[6] K.W.Hirsh and J.J.Tree, "Word association norms for two cohorts of British adults", *Journal of Neurolinguistics*, 14(???):1-44, 2001

[7] W.Labov, "The boundaries of words and their meaning", in C.-J.N.Bailey and R.W Shuy, editors, *New ways of analyzing variation of English*, pages 340-373, Georgetown Press, 1973

[8] D.L.Nelson, C.L.McEvoy, and T.A.Schreiber, *The University of Sourth Florida word association, rhyme and word fragment norms*, Technical Report ???, University of South Florida, Aug. 1999

[9] C.A.Nickerson and D.S.Cartwright, *An empirical thesaurus: Meaning norms for 90 common words, complete tables*, Technical Report 85, University of Colorado at Boulder, Oct. 1979

Be Assertive

Another really helpful tool is the use of assertions. Filling your code with assertions serves double duty:

1. If the assertion is violated, you've just found a bug.
2. The assertion provides additional documentation to the maintainer. e.g.

```
void foo(int i) {
  ASSERT(i>0);
  ASSERT(i<maxVal);
  bar(i);
  baz(someArray[i-1]);
}
```

Here, the assertions tell the maintainer about the allowed range of values for `i`, thus enabling him to reason better about the code, and verify that the access to `someArray` won't go out of bounds, for example.

The asserts shown in the example, are using asserts to define a contract. This is one of the tools used by the Design By Contract technique of defining pre-conditions and post-conditions for every function, which are then rigidly enforced. Programming languages such as Eiffel include support for DBC as part of the language, but other languages rely on assertions.

Improving existing code

Code isn't always in the form you might desire, even if you intended it to be, and it started out that way; requirements changes, bug fixes and new features often require modifications that aren't entirely within the spirit of the existing design, so the code starts to get ugly. Left unchecked, code can get very ugly indeed, so how can we get it back under control? The answer is, of course, *Refactoring* – changing the code in ways that improve the structure and maintainability of the code without changing its behaviour.

Ideally, you do refactoring in small steps, as you develop the code, with a comprehensive suite of automatic tests to ensure you don't break anything. However, you can refactor any code that's less than ideal, even if it is old and large and very ugly, with no automatic tests; you just have to do it in small steps.

Firstly, it is very important that you don't add new functionality whilst refactoring, since this just leads to confusion. Refactor the code first to make the new functionality easy to add, then refactor afterwards to tidy up, but don't refactor at the same time.

Secondly, refactor in small steps; take a long function and break it down into smaller functions one at a time for example. If you have automatic tests, run them after every change to ensure you haven't broken anything. If you don't have automatic tests, consider adding them, and then refactoring; you definitely want to be extra careful in this scenario, since it is not immediately obvious if you've broken something – was the function you just renamed virtual? Did you rename the function in the base class or derived classes as appropriate?

There are many resources available on refactoring, not least Martin Fowler's book.

Testing, Testing, 1, 2, 3

As mentioned above, refactoring is easier if you've got automatic tests, and most code needs refactoring as new features are added, or bugs are fixed, in order to keep it tidy.

One way to ensure you have a comprehensive suite of automatic tests is to write the tests *first*. Don't write a line of production code until you have a test to verify it does what is expected. Indeed, that is how you identify what is expected – you think of something the code should do, and write a test to show how it should happen from the client point of view, *then* you write the code that does it. This often means you are writing tests for non-existent classes and functions, because you haven't written them yet.

Such a technique is called Test-First Development, or Test-Driven Development, and is often advocated by agile development methodologies; it is one of the core practices of eXtreme Programming (XP), for example. This also involves step-wise design – your design evolves as you think through what is required for each test, rather than designing the whole architecture up front. If your test requires functions or classes that aren't there, or access to data that isn't directly available, then you add the missing features to make the test pass, and then refactor the code so it's tidier. You should never have more than one failing test at a time unless you're feeling really brave.

There are two key benefits to this technique: you have a comprehensive test suite for your code, so the chances of unwittingly adding a bug when you make a modification are small, and as a consequence of this, you gain more confidence when refactoring so it is easier to keep the code tidy, well factored and maintainable.

Of course, writing tests first isn't the only way to get a comprehensive automatic test suite, but it is certainly the easiest way I've found – it doesn't actually add that much time to the development process, and you're less likely to forget. It also ensures a high degree of coverage, if you don't write any code not required by the test, which is hard to achieve if you write tests afterwards – not least because the code has been written with testing in mind, so is inherently easier to test. Anyone who has tried to add tests to legacy code knows how difficult it can be, due to the interdependencies between things.

If you write the tests first, it is also hard to skimp on tests when the deadline looms and the pressure is on, since you already have them written. If I had a penny every time a project had its testing time cut or dropped altogether due to schedule pressure, I would be a rich man. If you're writing tests after the code, it is hard to have the discipline to ensure they are all written with sufficient coverage. Also, if you are writing tests for code that you've already written, and which you feel "works", then it feels more like drudgery than writing the tests first.

Refactoring in small steps and testing often has an additional advantage: the code is *always* in a releasable state, since you are never more than a few changes away from a version that passes all the tests. It might be that the code is half one design and half another, as you are midway through a large refactoring (one small step at a time), but because all the tests pass you know the application works as far as it goes, so you can release it easily. If you make large changes before testing, or allow multiple tests to be broken for a while, then you either have a lot of work to finish, or a lot of work to undo before you can get to a releasable state.

Keep it Simple, Stupid!

Simple things are easier to maintain. That means using simple algorithms and facilities until it is proven that a more complex one is necessary – don't use a relational database, when a simple text configuration file will suffice. It also means using the appropriate abstractions, and making good use of the available library facilities. Anyone who writes a sorting algorithm when using C++ had better have a jolly good reason not to use `std::sort` or `std::stable_sort`, for example.

Actually, writing simple code can be quite difficult; it requires a good domain knowledge to be able to choose appropriate abstractions, and a willingness to refactor continuously as your knowledge improves – both knowledge of the domain in general and of the customer's requirements in particular. It also requires discipline – it is very easy to over-engineer, with the idea "I'll need this later". A key idea that comes from the Agile methodologies is "You Aren't Gonna Need It" (YAGNI) – only add features and infrastructure if you actually need them for what you're currently doing, rather than in anticipation of future requirements. The chances are that when the future requirements come, if they come at all, they are sufficiently different from what you anticipated that the infrastructure you were going to build would have been insufficient, or addressed the wrong areas.

Conclusion

Writing maintainable code is not hard, it just requires careful thought. The key recommendation that I have is to Keep it Simple – most of the others flow from there, such as short functions, single responsibilities, well chosen names and doing things Once and Only Once. Refactoring is the means to keep things simple in the light of new requirements, and asserts and automated tests give you confidence that refactoring isn't going to break anything, whilst documenting how the system is to be used and constraints put on it.

Anthony Williams

References and Further Reading

Martin Fowler, *Refactoring: Improving the design of existing code*, Addison Wesley.

Dave Astels, *Test Driven Development: A Practical Guide*, Prentice Hall PTR.

`http://www.c2.com/cgi/wiki?YouArentGonnaNeedIt`

Reviews

Bookcase

Collated by Christopher Hill
<accubooks@progsol.co.uk>

A Note from Francis

I have had a couple of Linux users volunteer to help me with my next book but if anyone else would like to join in please do not be shy. Testing code for books is an important job and the more eyes that look at it the more likely it will work as intended when novices try it.

Only one person actually emailed me about my question in the last Bookshelf. Yes, it was the first one ever not to have a review from me in it.

Both Christopher and I are grateful for the number of you who have volunteered recently though there are still over 200 books sitting on my office stairs waiting for reviewers. The backlog is likely to have gone up between my writing this and your reading it because I will be away for almost a month from the time I ship this off to the people who need it.

Francis

The following bookshops actively support ACCU (the first three offer 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 me know so they can be added to the list

Computer Manuals (0121 706 6000)
www.computer-manuals.co.uk
Holborn Books Ltd (020 7831 0022)
www.holbornbooks.co.uk
Blackwell's Bookshop, Oxford (01865 792792)
blackwells.extra@blackwell.co.uk
Modern Book Company (020 7402 9176)
books@mbc.sonnet.co.uk

An asterisk against the publisher of a book in the book details indicates that Computer Manuals provided the book for review (not the publisher.) N.B. an asterisk after a price indicates that may be a small VAT element to add.

The mysterious number in parentheses that occurs after the price of most books shows the dollar pound conversion rate where known. I consider a rate of 1.48 or better as appropriate (in a context where the true rate hovers around 1.63). I consider any rate below 1.32 as being sufficiently poor to merit complaint to the publisher.

C & C++

C Programming in Easy Steps by Mike McGrath (1-84078-203-X), Computer Step*, 192pp @ £10-99
reviewed by Francis Glassborow

I was not intending to review any more books aimed at newcomers because having a book of my own aimed at such readers might be considered as a conflict of interest. However when this book crossed my desk I reached the conclusion that I really could not burden anyone else with it.

The publisher has done a great job of presentation. The book is a full colour publication on slick paper and at an exceptional price for books on programming. That is where it stops.

The front cover claims 'Plain English', 'Fully Illustrated', 'Easy to Follow'. Well it depends on your concept of what those terms should mean. The author does not appear to have any kind of concept of what is reasonable to expect of a newcomer. He often uses 'programming jargon' (i.e. terms with special meanings in a programming context) without giving any kind of explanation. 'Fully Illustrated' apparently means lots of marginal icons and endless screen shots of source code and output from executing toy programs. 'Easy to Follow' means that the author will tell you lots of things without showing you what they are good for.

The book does not have an introduction so I started to read the first chapter in an attempt to discover who this book is for. There I found a section headed 'Why learn C programming?' Here are the first two paragraphs of that section:

> The C language has been around for quite some time and has seen the introduction of newer programming languages like Java, C++ and C#. Many of these new languages are derived, at least in part, from C, but are much larger in size. The more compact C is better to start out in programming because it is simpler to learn.
>
> It is easier to move on to learn the newer languages once the principles of C programming have been grasped. For instance, C++ is an extension of C and can be difficult to learn unless you have mastered C programming first.

Leaving aside the somewhat weird use of English as manifested by the last sentence of the first paragraph (remember, 'Plain English' and 'Easy to Follow') the thrust of these paragraphs is quite untrue. C is a very difficult language for newcomers to programming because it is relatively terse and assumes familiarity with programming. That is not a fault with C, it was designed by a programmer for programmers and those who were already well familiar with programming concepts. C is sometimes described as a high level portable assembler. By its very nature it needs the programmer to understand something about the underlying hardware. I do not think anyone who has actually successfully taught programming would elect to use C as a starting point. Well I might in a classroom full of students who were well versed in electronics and wanted to write code for small, embedded systems but that is a very specialist group.

The very next section, 'Standard C libraries' starts with the following paragraph:

> ANSI C defines a number of standard libraries that contain tried and tested functions which can be used in your own C programs. The libraries are known as "'header files" and each have the file extension of' .h". The names of the C standard library header files are listed in the table below with a description of their purpose.

Note that this is the top of the second page of text. At this stage the reader has little if any idea as to what a function is and why it might be helpful. The reader is going to find it even harder to understand why s/he needs to tell the linker where the libraries are having been told they are header files (which they will later be told to #include into their program – well actually, no, because the author never invites the reader to write a program, just to copy his.)

Chapter One ends with a section on comments (I have passed over several further pages of plain geek speak, i.e. it is English but will not add much to the newcomers understanding unless they are already firmly immersed in the computer culture.). Here the author excels himself with a set of comments that demonstrate exactly what comments should no be. Things like:

```
/* declare the obligatory
main() function */
```

By the way he has no idea about the difference between 'declare' and 'define' which will make it harder for the reader when they come across code that separates out pure declarations (such as the main contents of a header file) from the definitions used to create object or library files.

Chapter two is about storing data. In 16 pages the author goes from 'Creating Variables' through such things as register variables and data conversion before finishing up with multi-dimensional arrays (note that we do not yet know what a function is.) Would you like to guess what he uses for data input? Yes, good old `scanf()`, just what every newcomer needs in order to reformat their graphics card (seriously, I once managed to do exactly that). Of course we need input, and if we are going to play with numbers we might even use `scanf()` but the problems a newcomer is likely to face deserve more than half a page of text, followed by a miniscule program and a pretty picture of the screen output from running the program.

These are not easy steps but whopping great leaps where being on the Moon would be a great advantage.

The next chapter is titled 'Setting constant values'. The first page covers the use of `const` in declarations (and the author comments, *'Notice that it is convention (sic) to use uppercase characters for constant names, to readily distinguish them from variable names in the program code.'* thereby ensuring that the poor innocent will get bitten by the pre-processor as often as possible.) On the next page the author covers `enum`s in four brief paragraphs.

The chapter finishes with three pages on the pre-processor (defining constants, conditional definitions and debugging definitions) and the poor reader has not yet learnt about functions.

In fact we have two more whole chapters ('Performing operations' and 'Making statements') to go before we get to functions. Let me skip those. At the top of the second full page of chapter six ('Using functions') we see the following code:

```
#include <stdio.h>
void first();   /*declare
function prototypes*/
int square5();
int cube5();
```

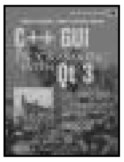
Now remember this is C, not C++ so that comment is entirely wrong. Those are not prototypes because prototyping a function with an empty parameter list in C requires the use of `void`. However this page does lead me to apologise for an earlier insult. The author does know the correct use of 'definition' as applied to a function, it is just sad that he did not use it earlier on.

I think I have gone far enough. A reader who half way through this book can only cope with a simplistic function definition such as:

```
int square5() {
  int square = 5*5;
  return square;
}
```

is not going to cope with chapter seven on bitwise operators.

Frankly I would only recommend this book to someone whom I wanted to persuade to never ever try to program. An experienced programmer wanting to learn C would use something such as K&R, someone with no programming background would be much better off learning about programming using a language which is considerably easier than C.



C++ GUI Programming with Qt3 by Jasmin Blanchette & Mark Summerfield (0 13 124072 2), Prentice Hall, pp @ £35-99 (1.25)
reviewed by Paul F. Johnson

Just like you know when you have a really dire book, you definitely know when you have an excellent book and this one is definitely an excellent one.

It starts off with the usual "Hello World" using straightforward, uncomplicated language. Everything is explained and explained in enough detail to convey not only what happens on the surface, but also slightly deeper down. This "taking by the hand" approach is incredibly effective and with the clear graphics and writing style demonstrates how to create applications quickly and efficiently.

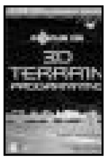
As you go further into the book, the level of complexity increases, but at the same time, so does the explanation.

All of the main classes are covered – windows, menus, messages, slots and signals and covered in understandable language.

The book comes with a CD containing all of the source code as well as Qt 3.2 for Windows (non-commercial licence – in itself worth more than the value of the book), Linux and MacOS. It also comes with Borland C++ 5 (non-commercial) and Borland C++ 6 (trial version).

With Qt being one of the main widget sets on Linux (due to the amount it is used with KDE) as well as growing in popularity on the Windows and MacOSX platforms, this book is not only great value, but a very good way into learning about one of the premier cross platform widget libraries.

Okay, it's not much use if you don't understand C++, but then if you don't understand C++, why would you buy this book? Highly recommended.



Focus on 3D Terrain Programming by Trent Polack (1-59200-028-2), Premier Press*, 218pp @ £21-99 (1.36)
reviewed by Francis Glassborow

I looked long and hard at this book before deciding to review it. There are three different aspects that I wish to touch on; who is it for, what does it cover and how good is it technically.

Here are a couple of quotes from the books single page Introduction:

> This book will take you from novice programmer with no terrain programming knowledge at all to a fully informed terrain programmer who can implement some of the most complex algorithms around. What exactly is this book about? Well, I'll you.

and

> This book moves at a fast pace, but nothing that any C/C++ experienced programmer with some slight knowledge of basic 3D theory will have trouble understanding.

The second quote makes it clear that the author is not addressing novice programmers but programmers who are novices in the problem domain. It also suggests that he lacks a clear idea about what is C and what is C++.

That last sentence of the first quote shows just how little care his publisher took. My copy editor and proofreader would never have let me so embarrass myself so early in a book.

It comes as no surprise to discover that the author is still in High School. I give him all credit for attempting a book such as this one, but he needs to select his publisher with more care because he needed much more support than he clearly got. All authors need good editing teams giving support, but young, technical enthusiasts need it more than most.

The book covers various programming algorithms to deal with terrain generation. Good, reasonably realistic, terrain is needed in many modern games. This means that those who aspire to trying their hands at game programming need a book such as this one. As far as I can see (and this is not a problem domain in which I have much expertise) the author covers a range of suitable algorithms with sufficient detail so that those willing to work diligently will come to understand them.

However the quality of the source code is much what we might expect from an enthusiastic but largely self taught young programmer. The class design leaves much to be desired and the low-level implementation code is a confused mish-mash of C and C++. Perhaps this can be turned to advantage in that the diligent reader with any genuine programming expertise will wish to redesign the code and develop a quality implementation. There will be no intellectual property right issues because the code will be brand new.

So who might get good value from this book? Well you need to be a confident and competent C++ programmer (or a programmer in some other language who can read poor quality C++ and re-implement the ideas.)

What disappoints me is the contribution made by the series editor, André LaMothe, who should be doing more to ensure that young contributors to this series are given proper support.

Visual C++ for Visual Basic Developers by Bill Locke (0 672 32218 8), Addison-Wesley, 420pp @ £36-50 (1.37)
reviewed by Frances Buontempo

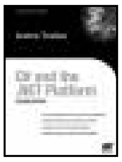
This book briefly introduces C, C++, and C# comparing their syntax and keywords with Visual Basic. Clearly, a 420-page book can only give some initial pointers to these three languages. Its aim is to allow VB developers to write C DLLs and ActiveX controls using the ATL with Microsoft's Visual Studios 6 and 7, as well as giving a small summary of C#. It succeeds in this aim.

However, the lack of details may prove frustrating for the reader. The first chapter provides a history of VB. The C chapters focus on problems with casting between numeric types, give details on various pre-processor directives that may put off a novice and show how to build a C DLL that can be used in VB. Nicely, however, code fragments comparing VB declarations of basic types are given side by side with C declarations to familiarise the reader with keywords and syntax.

The C++ chapters introduce classes and function overloading. Since no mention is made of templates, the section on using the Active Template Library is simply written on a need to know basis, and so the code listings contain many elements unfamiliar to a reader who only knows VB. In addition, it is not until these chapters that we get a whole code fragments that will compile and do something independently (rather than needing client code). Most include void main, rather than int main, and one uses printf to write an output - something that would be better placed in the C chapters, rather than a quarter of the way through the book.

This book is OK and will allow a VB developer to write some add-ins in C and C++, as well as giving an overview of C#. However, it only provides a brief introduction, and further reading would be required for a more thorough understanding.

C# and Java



C# and the .NET Platform 2ed by Andrew Troelsen (1-59059-055-4), Apress, 1158pp @ £43-00 (1.40)
reviewed by David Sullivan

This is the second edition of the book, updated to account for the .NET changes that have taken place since the first edition in 2001. It covers .NET 1.1 and 2003 version of C#.

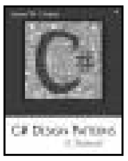
The book is a weighty tome of 1100 pages. It is substantial in subject matter also. It comprises of five sections; Introduction to .NET and C#, The C# Programming Language, Programming with .NET, Leveraging the .NET libraries, and Web Applications and XML Web Services. Troelsen does a very competent job explaining the detailed aspects of both C# and the .NET framework.

Not only are the concepts described here; there is a pragmatic aspect also. For example, garbage collection is treated in detail, including potential pitfalls of using unmanaged resources and how to override the default `Finalize()` method to clean up such resources *[CMH: but see Alan Griffiths's paper "Heretical Java*

#1" in Overload 59, which points out that Finalize may NEVER be called…]. The examples are pithy, but illustrate the text. The author uses a theme of automobiles in all the examples for simplicity. Not all will warm to this topic but one does not need mechanical expertise to follow the examples.

Just some of the aspects are Windows forms programming, graphics, ASP.NET and web development, ADO database, Type Reflection, and Object Serialization. All covered in sufficient detail to warrant this book as a serious contender for the book you are most likely to turn to if you need to refer to a C# or .NET topic.

This is not a beginner's book. The author has not considered the needs of a novice programmer. But if the potential reader has some programming experience this book is an excellent and thorough reference on the subject.



C# Design Patterns: A Tutorial by James W. Cooper (0-201-84453-2), Addison-Wesley, 390Pp+CD @ £34-99 (1.29)
reviewed by Patrick De Ridder

Cooper's C# Design Patterns is a cookbook of solutions to design problems, using Visual C#.NET. Knowing your informational requirement, you would want to find the recipe quickly, and the instructions to be clear. Cooper's book isn't easily used as a catalogue of design solutions.

Although the index of the book says otherwise, I find the book to be in three parts. Part I is an excellent one hundred page C# summary and reference. Part II is about constructing user interfaces, using the .NET Framework. Part III is more about design issues per se. By starting off with an introduction to the C# language, Cooper targets his book at C# novices.

Parts II and III of the book consist of chapters per design pattern, and one example or more, plus its code, per chapter. Much of the text in the chapters is about the examples used. All the example code is on the accompanying CD. Snippets of it are repeated in the chapters, and are discussed at the level of code lines. The reader must shuttle back and forth between the book and the CD code. However much novices will learn by searching through the example code, this book could do with less elaborate examples that more readily and effectively convey the design principles. Although Cooper has an intelligent and pleasant style of writing, Parts II and II of his book cannot be readily used for reference. The design patterns are not presented in a way that allows the reader to quickly identify a solution to a particular design problem.

If you were to buy the book just for its C# summary and reference, that alone would be money well spent. As to the rest of the book, it is Visual C#.NET oriented and not just about C#, as the title suggests. The reader might want to study the book in sufficient detail to be able to write a top level overview and, after finding a match between a particular design query and a design pattern in the book, continue to study the relevant chapter in further detail. Sitting down and reading the book from cover to cover is not what I would suggest, but who would do that with any cookbook?

Head First Java by Kathy Sierra and Bert Bates (0-596-00465-6), O'Reilly, 617pp @ £28-50 (1.40)
reviewed by James Roberts

This is a book that is aimed at teaching Java to a reader who has no knowledge of Java, but has some (but probably not a huge amount of) programming experience. It works very hard at explaining various details in a variety of ways that are designed to help the reader's memory; including humorous cartoons, doggerel, diagrams, 'interviews' with classes, annotated code extracts and a variety of exercises, which the authors strongly encourage the reader to complete. I am not a teaching expert, but I felt that I could use the techniques used by the authors in my own training courses (largely based around ways of saying the same thing in 3 different ways in quick succession without becoming dull).

In general the book works well. I found it interesting in the way that it presented Java in a not overtly technical manner; the prose was readable and generally well structured. For example its coverage of object references I thought was well done remaining accurate while being clear to a non-expert reader.

My only grouses with the book were minor typos, and the claim that you don't need to know the operator precedence rules in Java if you code with plenty of brackets.

However, these are very minor complaints and I would still recommend this book for a newcomer to Java to get started in the language.

Java 2 Certification Test Centre by (0 7821 3008 9), Sybex, ?pp @ £36.99 (1.35)
reviewed by James Gordon

Well this is the first time I've done a "Book Review" on a piece of software...

The application allows you to run ten tests in either practice or timed mode. You can set it up to give immediate feedback as you answer each question and it also explains which answers are correct and why, referencing back to a Sybex Java 2 certification book's chapter for more information.

Each test consists of 65 questions, but as you are doing the test you have no idea which question you are on.

After the test it gives you a total score and a score per each section of the test. This allows you to see where your strengths and weaknesses are, enabling you to focus your learning in specific areas.

You can review your answers for each test, seeing where you went wrong. Again my only gripe in this area is the list of questions only shows the first few characters of each question (about 20) and so it is hard to see what the question was. Also the list doesn't distinguish the questions you got right or wrong. For that you have to select the question.

All in all a very good application, but the only thing that distinguishes it from something you can download/run from the net is the history over time of your tests. Whether that is worth £36 is entirely up to you.

Mastering Java 2 J2SE 1.4 by John Zukowski (0 7821 4022 X), Sybex, 900Pp+CD @ £37-99 (1.32)
reviewed by James Gordon

What a tome of a book, but it would have to be to encompass the whole of j2sdk 1.4.

This seems a very good reference book for the SDK with a nice touch being a list of chapters where the new features for 1.4 are discussed.

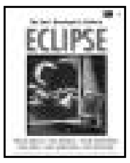
Each chapter is well laid out with pictures of the output of the sample programs. These sample programs are small and self-contained. They are also complete rather than being snippets of code that don't compile by themselves.

There are a lot of lists of methods and variables from classes and interfaces; this is good from my point of view as I'm trying to use this book to get my SCJP certification. It lists the method and a brief description of it functionality.

The book starts from the basics of Java and then goes into detail about all of the j2sdk features including swing, streams, beans, applets and printing.

It contains a CD with the entire source for the book, j2sdk 1.4.0, JBuilder 6, Optimizeit, Tomcat, Ant, JUnit, JIndent and the BDK.

An excellent book; huge but complete.



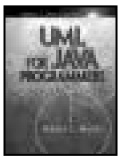
The Java Developer's Guide to Eclipse by Sherry Shavor et al. (0-321-15964-0), Addison-Wesley*, 854pp+CD @ £37-99 (1.32)
reviewed by Silvia de Beer

Some, like me, tempted to buy this book by just reading the title, might be disappointed. The book is divided in three parts. The first part titled Running Eclipse, the second part, starting on page 185 about Extending Eclipse, and the third part, starting on page 623, filled with exercises. Together with a verbose style of writing this fills up the 854 pages.

If you consider the most important part of this book the second part (about extending Eclipse) the book's title should be interpreted as "The Developer's Guide to extend Eclipse". The book's title should definitely not be interpreted as "I am a Java developer, how to use the Eclipse IDE to write my Java code". The first part is not extensive enough to serve as more than an introduction to how to use Eclipse.

The second part of the book is slightly more useful, but does not go into the details that you need if you really want to extend Eclipse. If you want to extend Eclipse in a standard way, not wanting too much functionality, this will be clearly described in the second part. However if you want to step slightly besides basic functionality, you will probably have to struggle your way through the Eclipse API documentation and mailing lists. The third part offers you some basic exercises for all the chapters of part I, and one basic hello world type exercise covering the whole of part II. For the rest, each chapter in part II directs towards example code on the CD, which is interesting, but nothing much more than what can be downloaded from the Eclipse web site as well.

The book comes with a CD with Eclipse 2.0, handy given the size of the software, but not so handy if you consider that very soon the first release of 3.0 will come out. The CD also contains the exercises.



UML For Java Programmers by Robert C. Martin. (0-13-142848-9), Prentice Hall, 249pp @ £31-99 (1.25)
reviewed by Emma Willis

I have thought hard about this book; I can't decide whether it is a clear and concise demonstration of common-sense design principles, or whether, in its attempts to preach the simplicity of UML and OO design, it fails to be of any use in teaching the principles it intends to illustrate.

The book's name suggested an introduction to the Unified Modelling Language. Unfortunately, if you were new to UML, having read this book you would come away with few new skills, and a very patchy knowledge of UML diagrams.

I enjoyed reading the author's common sense, real-life examples of how, and when to use the different types of UML diagram. The more complex, detailed nitty-gritty of UML was glossed over, and in many cases dismissed altogether as unnecessary.

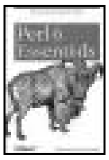
However, after chapters on class, sequence and use-case diagrams, I began to get a little annoyed at the author's attitude to UML. I wondered why some elements were so heavily criticised, and I questioned why the author had decided to write a book on UML if he believed so little of it was actually useful.

At this point the book veered off to cover some core principles of Object Oriented Design (the author's speciality), of extreme Programming and of component packaging principles.

The closing chapters successfully demonstrate the use of all of the UML concepts previously covered. You are challenged to design a Coffee Maker, and then shown a well-designed, OO solution using sequence, class and object diagrams, highlighting common mistakes made when faced with this challenge.

This book begins with the right goals, to demonstrate the simplicity of OO design with UML, and to remove some of the mystery of UML by only focussing on the elements that designers and developers are likely to use on an every-day basis. However, it would perhaps be better described as an introduction to object-oriented design principles, using some UML and some unified process practices.

Other Languages



Perl 6 Essentials by Allison Randal, Dan Sugalski & Leopold Totsch (0-596-00499-0), O'Reilly, 194pp @ £17-50 (1.43)
reviewed by Frank Antonsen

Something exciting is taking place in the Perl world. The language has grown enormously. And although this has made Perl an extremely flexible programming language, it has also made it very hard for developers to work on maintaining the Perl interpreter/compiler system. The Perl core has become too complex.

This has led to the launch of Perl6 – the next major release of the language. This is more than a new release; it is a complete rewrite of Perl's internals, and a major overhaul of the syntax and structure of the language.

To make this as flexible and powerful as possible, it was decided to re-build Perl on top a virtual machine nick-named Parrot after an April Fool's Joke on the O'Reilly web page a few years back. Like Microsoft's VM for the .NET project, Parrot is going to be language neutral, and is designed from the start to be able to accommodate Python and Ruby besides, of course, Perl6 (and Perl5). This VM can be programmed in PASM (Parrot Assembly language) or PIR (Parrot Intermediate language) handled by the IMCC (Intermediate Code Compiler). These are both very high-level abstract assembly languages, which are then translated into Parrot bytecodes.

However, the Perl6 language itself has not been designed yet. This is done through a number of so-called Apocalypses put out by the creator of Perl, Larry Wall. These sum up the various proposals made by Perl users and combine them to a coherent new language. Thus, rather little can be said about this future programming language, except that it will add a number of very powerful concepts, such as continuations, co-routines and multi-methods. But little can be said about how these are to be implemented yet, and the book only gives a brief coverage of them.

As a result, a lot of the information in this book is tentative. A new release of the book is planned every year until Perl6 finally comes out. In the mean time, it serves as an excellent overview of the process and where Perl6 stands. For the latest information, one has to look at the actual code and documentation of it. The book contains plenty of information on how to get this code, how far the project is, and how to participate. It is a very courageous attempt to capture the present state of this highly elusive target.

The book is very well written, with only a few flaws. Mostly, these flaws are inevitable and follow directly from the elusiveness of the Perl6 language. Occasionally, though, one can see that the book is the result of the collaboration of three authors. For instance, the concept of PMCs (Parrot Magic Cookies – the way objects and other kinds of complex data structures are handled) is mentioned quite early, but they are not defined until chapter 6. These flaws do not distract from a very favourable impression, however. Highly recommended.

Core PHP Programming, 3ed by Leon Atkinson (0-13-046346-9), Prentice Hall, 1067pp @ £35-99 (1.25)
reviewed by Tim Pushman

Core PHP is quite a large book, with over a thousand pages. It is part of Prentice Hall's "Core..." series, which generally have a high level of quality, and this book is no exception. The production quality is good, the writing style is clear and there are very few typo's. Leon Atkinson has the knack of writing in short clear sentences, yet still maintaining readability.

The book can be considered in three main sections. The first section of about 160 pages covers the basics of PHP, the core language and structures and the rudiments of network connectivity. The second section is the main reference work, of about 770 pages and the final section, of about 90 pages covers general software engineering with PHP.

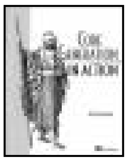
The meat of the book is really the middle section, which taken on its own would be a good reference work. The first and third sections cover such different audiences that it distracts from the focus of the book. Is it a book for beginners, a reference work or "PHP Best Practices"? Given its size it may have been better to have made two books out of it, one as a reference, one as a programmers guide. The size of the book makes it unwieldy for quickly looking up information.

My main complaint though is that the reference section does not differentiate between functions that have been introduced in PHP 5 and functionality that existed before then. This is important in a reference work, especially as PHP 5 will not be in general use for another year or so. The online PHP references all indicate which version of PHP first used a particular function and it would have been easy to do the same here. I can only guess that the intent is to spur people on to using PHP 5, but it does detract from the usefulness of the reference.

PHP 5 is certainly a major step forward for PHP, which is already a robust and well-established language on the internet. Unfortunately, as it is very well established, most ISPs and hosting companies are going to hesitate before doing a major upgrade to PHP 5, which is why I feel that it may be a year or so until it is in common use.

Reading the book certainly gives an exciting taste of the future of PHP and, although the book could have been better, it still gets a rating of Recommended. It won't be the first book you buy for learning PHP, but it could be the second.

Other Programming



Code Generation in Action by Jack Herrington (1-930110-97-9), Manning, 342pp @ $44.99
reviewed by Mathew Davies

Code generation involves writing programs to generate the source code for other programs. This book demonstrates how it can sometimes make good engineering sense to generate source code automatically. The author explains what code generation is, when automatic code generation might be appropriate, how it might be accommodated within the software development process, and how you might go about organising and managing code generation.

Mr Herrington chooses the Ruby programming language to build his code generators, using XML to build the text files that drive his generators. He demonstrates how combining the Ruby programming language with its ReXML extension provides a means of extracting all necessary information from these input files.

The book demonstrates the code generation technique by showing you how to build a series of small but useful generators. A particularly nice touch to this book is the author's attention to the practical details. For example, he makes time and space to discuss the types of objection that other members of the software development and management teams might raise to the use of code generation. He also provides you with advice on techniques for dealing with such objections.

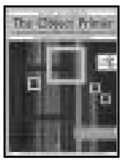
I feel that the real value of this book lies not in the particular examples that have been chosen to illustrate the technique but in the ideas that it generates for applications elsewhere. The book certainly seems to justify its current retail price of around £25.

The Power of Events by David Luckham (0 201 72789 7), Addison Wesley, 376pp @ £34-99 (1.29)
reviewed by Lawrence Dack

This book is set in the context of distributed systems and message-based protocols. Its objective is to introduce and describe tools and techniques for analysing complex events, where an 'event' is a basic transaction (the transmission of a message from one system to another) and a 'complex event' is the occurrence of a set of events which possess a higher meaning. The premise of the book is that, while low-level analysers exist which provide a view of the basic events occurring over a network, the view they provide is not sufficient for an understanding, far less a debugging, of system behaviour. Complex Event Processing (CEP) is introduced as a means to provide that understanding by detecting patterns of lower-level, basic events. This seen as the key to providing system viewers and analysers that are flexible, hierarchical and customisable; providing individual views of a complex system at what ever level is appropriate for the task in hand.

The book is presented in two very different parts. Part one introduces the concept of complex events, their nature, their potential uses, and a simple declarative language for expressing rules to recognise complex events, and actions to take when one is detected. It is written in a readable style, reasonably free of jargon, and serves as an introduction of the topic to an intelligent layman. Part two is much harder going. Here the author describes Rapide, an event processing language developed over ten years of research at Stanford University, and shows how it can be used to solve some of the problems discussed in Part one. The description of Rapide's use is very thorough, almost at the tutorial level. The final chapter looks forward to discuss how a CEP application (i.e. a tool like Rapide) might be implemented using commercial technology. After describing the main features that a CEP application would require, the author concludes that the requirements are out of reach of current technology – but not for long.

Because the CEP concept presented is not immediately realisable, I assume the authors aim in writing this book was primarily educational: to highlight a growing problem, suggest an approach to solving it, and back this suggestion up with a distillation of 10 years research. I think he has largely succeeded in this. The question left in my mind is whether the education repays the intellectual effort needed to digest the books contents, particularly part two – and for me the jury is still out on that one.



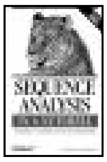
The Object Primer 2ed by Scott Ambler (0 521 78519 7), CUP/SIGS, 523pp @ £27-95 (1.43)
reviewed by Greg Billington

This book certainly packs a lot of topics into its 500 pages, covering quite a breadth of OO and described in a logical sequence from beginning to end of the lifecycle, including the development process itself.

The book describes itself as a primer and it is certainly easy to read, with a clear layout. The page layout includes greyed boxes of Tips and Definitions and there is a liberal sprinkling of diagrams throughout. Initially I found that certain items seemed to be repeating but this grew on me as I found it a book that could be read in sections and put down. When I picked it back up a definition of a term was never more than a page or two away, and listing only terms relevant to the topic area making it easy to see the relationship between certain words. There is a standard glossary at the back of the book but these definition boxes help the student to OO and make the book appear less formal and more a teaching material, which it is. Each chapter ends with a short summary, typically only a paragraph, which really is very high level before a few questions for review purposes. The book has lots of examples, some of which seem very wordy and some slightly humorous, which I was unable to decide if this was good thing in keeping the book light and readable or if it seemed to be talking down to me.

As for coverage it starts with gathering requirements and validating them then moves onto OO concepts. The biggest sections are analysis then design; the programming section covers how to translate the design into Java, with lots of Java snippets. Each of these chapters includes not only an explanation of the topic but steps to follow and has the authors advice in the forms of tips, some made me nod in agreement whereas others seemed to have too much preaching and not enough background, aimed at a novice student they are OK but they become tiring for anyone more advanced.

Overall an introduction to OO software in structured steps written in a teaching style, easy to read for a novice.



Sequence Analysis in a Nutshell by Scott Markel and Darryl Leon (0-596-00494-X), O'Reilly, 286pp @ £20-95 (1.43)
reviewed by Ivan Uemlianin

This book is a digest of data format and command-line parameter summaries for a selection of bioinformatics software packages. Its intended audience is working bioinformaticians already familiar with the software. For this audience I can recommend it, as the online documentation that comes with the tools is often lacking. The book is *not* introductory or tutorial in any sense, excepting that it's easier to browse through a 286 page paperback than it is to wade through a handful of help files.

The book is in three parts: Data Formats, Tools & "Appendixes".

Part 1 covers FASTA, GenBank/DDBJ, and EMBL DNA formats, and protein formats SWISS-PROT, Pfam and PRO-SITE. Part 2 covers: Readseq; BLAST, BLAT, and ClustalW; HMMER; MEME & MAST; and the European Molecular Biology Open Software Suite (EMBOSS, which gets 170 pages). The Appendices cover background material including DNA and amino acid codes, and genetic codes for various organisms. (See the longer review on the web for more discussion, but if you don't know what any of this means, this is not the book for you.)

This documentation is available elsewhere, but not readily, so the book is more than a bunch of man pages in perfect binding. The information is very terse but, if you're working frequently with these tools, that's probably exactly what you want. There is just enough annotation to prod your memory, but no more.

The authors, both practising bioinformaticians, have set out to produce a 'nutshell' book that they would use themselves, and this accounts for the book's strengths. It also accounts for its two main weaknesses. The narrowness of focus is the first of these, and if your needs are fulfilled by the focus, it may not be a weakness at all.

More seriously, there are some hidden gaps in coverage. Some examples: the Pfam chapter does not cover the actual sequence format (i.e. only the #=GF section is covered, not the #=GS & #=GC sections); the BLAST chapter covers only NCBI-BLAST (and not e.g. WU-BLAST); the EMBOSS chapter doesn't mention the EMBASSY range of open source programs.

The appendices might have been cut & pasted from the authors' own post-its. Citations are given, but the reproductions are perfunctory. If you know the sources this will not matter, but it doesn't make for readable documentation. The 'Resources' appendix is a mess. *Some* of the references from the rest of the book are given here, along with a mixture of introductory texts, specialist monographs and a few from O'Reilly.

Faults notwithstanding, the book fulfils its aim reasonably well – with some rough edges, and check the tools you use are included. On the whole, it's a good 'nutshell'. Recommended.

Methodologies

Domain-Driven Design: Tackling Complexity... by Eric Evans (3-321-12521-5), Addison-Wesley, 528pp @ £37-99 (1.32)
reviewed by Huw Lloyd

Domain-driven design is concerned with aligning the core of software systems with the domain they represent. To this end the software is used to express the domain model. This book is a unified a collection of recognised software pattern descriptions, supported with UML sketches and java code snippets, suited to communicate the major concerns of iterative domain design.

'Refactoring through deeper insight' is a theme that permeates the book, an agile approach advocated by Evans, to manage the complexity of software by a continuous process of discovery of domain knowledge and its representation. Through domain-driven design Evans' advocates iterative design and understanding. When a better understanding is achieved he encourages the developers to reflect that understanding in the software's design by refactoring.

The articulation of the patterns in conjunction with a good breakdown of key programming concepts provide an exceptionally clear presentation of software design with respect to alignment with the target domain. Each sub-topic is relevant, whilst the flow and structure of the whole book is maintained from building blocks through to large-scale structure.

I was rather surprised at the scarcity of references. Although the author refers to many influences 'in the air' during the incubation of this material, I would have expected to see reference to seminal works such as Jackson's 'Problem Frames' and Coplien's 'Multi-Paradigm Design for C++'. As it stands only a handful of books are mentioned and particular bias is given towards Martin Fowler who provides a forward for the book.

Irrespective of whether you feel agile software development is for you, I highly recommend this book as a source for insight into class, module and system design.

Balancing Agility and Discipline by Barry Boehm & Richard Tumer (0-321-18612-5), Addison Wesley, 266pp @ £22-99 (1.30)

reviewed by Pete Goodliffe

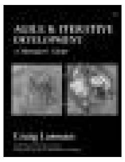
This is a timely and authoritative software development book from well-respected authors. It stands in the chasm between the 'agile' and 'disciplined' software development approaches and pulls the two together. The authors contrast and compare methodologies, and explain how elements of both can be tailored into your most appropriate development style.

The content is certainly nothing new - development processes have recognised this overlap and begun to span the divide already. However, the authors provide a clear pragmatic viewpoint, explain how to gain the most from each methodology, and describe how to create a specific 'balanced' development process for a particular company/project.

Appendixes summarise the key agile and disciplined methodologies, although the reader does need a good prior understanding to fully appreciate the information.

The book is thoughtfully laid out, following the novel approach of Taylor's 'Object Technology: A Manager's Guide' - margin notes contain a 'fast track' summary of the text.

This is a highly recommended book for software team leaders and development managers.



Agile and Iterative Development by Craig Larman (0-13-111155-8), Addison Wesley, 342pp @ £27-99 (1.25)

reviewed by James Roberts

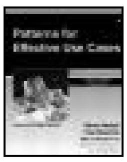
This book is a high-level overview of agile and iterative development styles. It provides an overview of the reasons for agile development, giving the motivation for using these methods, and some contrasting and complementary descriptions of important examples.

The book is at its best in its descriptions and comparisons of four agile methods (Scrum, XP, Evo and RUP). The descriptions are in enough detail to give a good flavour of each one, and how they should (and should not) be applied - with extensive cross-references to each other. This both contrasts the strong and weak points of each method, and shows their similarity, which gives a good intuitive background of how agile methods work.

Although the book does not give enough information on the agile methods to allow the reader to start using them in anger, the book includes recommended further reading with each chapter.

I found the book somewhat evangelical - giving the impression that no waterfall-based project could possibly succeed (backed with large amounts of rather one-sided case study evidence). This grated at times. I would have liked to see more negative experiences on the agile side - not just on the waterfall side.

In summary, this is a readable book, which gives a good start point for anyone interested in agile methods. The main reservation that I have is based on its evangelical tone - the author's clear enthusiasm for iterative development seemed at times to overshadow his critical facilities.



Patterns for Effective Use Cases by Steve Adolph et al. (0-201-72184-8), Addison Wesley, 236pp @ £26-99 (1.30)

reviewed by Lawrence Dack

The aim of this book is to provide software developers with "a source of objective criteria by which to judge quality and effectiveness [of use cases]". To this end, the book describes some three-dozen patterns, each of which describes a desirable characteristic. Perhaps half of these address issues of good use case writing practice. The scope of the rest is wide-ranging: some patterns apply to the use case development team, others to the process of writing use cases, still others to desirable properties of a use case collection. Despite the book's title, the 'pattern' concept – in the sense of a clearly identifiable template - is less evident than in the programming language patterns I have seen described: personally, I would describe these more as principles than patterns.

Nomenclature aside, the patterns described here encapsulate valuable advice in my own experience. I am at the tail end of a project to describe the requirements for complex information system. Some two-thirds of the patterns in this book were directly relevant to the project, and in almost all of these my experiences on the project lead me to agree with the authors' description of the issues and the conclusions reached. For me that endorsement is a significant testimonial to the quality of the content, and I would feel that the authors have achieved their stated aim.

However I do have reservations about the manner in which the advice is presented. This is a shame, because the authors have obviously gone to some trouble to make the book accessible: unfortunately, I didn't appreciate some of the devices employed – for example, fictitious anecdotes with 'you' as the wise consultant I found to be more of a distraction than a clarification. This should not detract from the basic worth of the book – it just makes that worth a little harder to extract than it should be.

Software Configuration Management Patterns by Stephen P. Berczuk/Brad Appleton (0-201-74117-2), Addison-Wesley, 218pp @ £30-99 (1.29)

reviewed by John Kewley

This is one of Addison-Wesley's Software Patterns Series and contains information on sixteen patterns in the area of Software Configuration Management (SCM).

This book provides excellent coverage of the most commonly applicable SCM patterns for both codeline and workspace management along with some universally applicable testing patterns.

Unlike some previous books in this field, the book clearly concentrates on describing pragmatic configuration management essentials for software developers, as opposed to managers. The result is an excellent starting point for agile, and more formal, software development methodologies. Note that patterns for documentation, authorisation and other CM policies are not gone into in any depth, however these are aspects of CM as a whole and peculiar to SCM; they are also already well covered in covered elsewhere.

The first three chapters of the book set the scene by providing a lot of background information on how SCM relates to software development and team. It then gives an introduction to patterns. The remainder of the book presents a catalogue of 16 SCM patterns (7 on codeline management, 6 on workspace management and 3 on testing). Each pattern is described in terms of the problem the pattern addresses, how it deals with the problem, when the pattern is applicable and any outstanding issues.

Finally, there are two appendices pointing the reader at online SCM resources and discussing current tool support for the concepts discussed.

Over the years, I have found that there are two important aspects to consider in a software system with respect to configuration management. The first is that you want sufficient control to avoid disasters, but not too much that software development is seriously impeded. I have been in too many projects in the past where a too strict or too loose codeline policy has been defined. This book clearly highlights the differences between different policies and is invaluable in this respect.

I would definitely recommend this book to anyone wanting to add a bit more rigour to their software development process, or as a handbook to help in the setup of the version control part of their process.

Software Engineering with Ada by Grady Booch (0-805306-04-8), Benjamin Cummings, 575pp
reviewed by Colin Paul Gloster

Grady Booch used to be actively involved with Ada before he ported the Booch components to C++ and merged his diagramming notation into UML. At the end of 1984 he stopped writing an Ada advice column in "Ada Letters" which had a very different style to this book but a similar target readership.

Page 63 is the first to contain any Ada code. The rather late appearance of code is deliberate so as to devote earlier pages to such issues as the software crisis; the history of Ada; software engineering and determining nouns to be mapped to objects. From page 404 onwards the chapters also have no code but contain less welcome discussions on IDEs and future trends.

Unlike what is unfortunately the majority of programmers, Grady Booch appealed against importing packages into direct visibility. Ada 95's `use type` mechanism is a comfortable way of enjoying some of the conciseness brought about by an Ada 83 `use` clause while not introducing any scope clashes. The book described an Ada 83 substitute of a `use type` clause. Something else which you might find in old code could be optimisations for `accept` statements recommended in the book. Though `accept` would be discouraged in one particular modern community, the author's dismissiveness of polling in favour of Ada tasking would have been unfair even when the book was written.

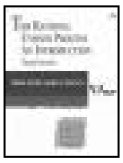
As this book is from before the effort to define Ada 9X, it contains claims that Ada 83 is OO. The object based decompositions in the book lack inheritance and polymorphism. For areas of his solutions where classes or abstract data types are not needed, Grady Booch had abstract-state machines: which he had much more easily represented by Ada packages than the insecure, inflexible awkwardness of the Singleton pattern in C++ in Gamma's; Helm's; Johnson's and Vlissides's "Design Patterns: Elements of Reusable Object-Oriented Software".

He provides an explanation for the language oddity which is the locating of a package's `private` section in the package specification instead of the `package body`. Unfortunately a few small mistakes and some misleading explanations for other language elements survive in the second edition. Very few spelling mistakes are to be found in the English sentences but illegal spaces are common in the code.

The blurb boasts that this edition has an "updated section on object-oriented techniques to reflect the current state of practice" without drawing attention to one huge `case` statement all of whose branches are identical. A default may be justified from time to time, but his assertion that `when others =>` "is useful" contradicts guidelines issued by a number of parties.

One of his examples has a memory leak. This latter edition somehow contains no warning of some of the quality issues with supposed Ada compilers of the time.

Generally Ada is correctly and easily explained. The book is designed to be sped through.



The Rational Unified Process: An Introduction 3ed by Philippe Kruchten (0 321 19770 4), Addison Wesley*, 310pp @ £26-99 (1.30)
reviewed by Emma Willis

I have not read any of the earlier editions of this book, so I can only assume that, like the software processes it describes, this book has now been through a number of iterations that have reviewed and refined its content. This book reads well – it is clear, concise and relevant; every chapter feels well thought out and thorough.

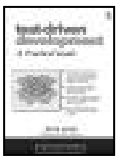
I had imagined this book to be a blatant advert for IBM's Rational products but was pleasantly surprised to find that the software tools themselves were only briefly mentioned as part of a tools summary towards the end of every chapter, explaining which products would be suitable for implementing and ensuring the success of the processes discussed.

The introductory chapters of the book summarise the disciplines involved in a unified process development cycle and place the disciplines in the context of an iterative model, and the RUP phases of development.

Later chapters go into more detail of the roles, artefacts and processes covered by each of the development disciplines such as Business Modelling, Implementation and Testing.

The book makes good use of UML workflow diagrams to explain all of the roles, artefacts and processes that it discusses; this tends to make each chapter come together. The book comes with a foldout poster of all of the disciplines that contains these workflow diagrams, and that places each discipline at the appropriate point in the phase/iteration model. I am not sure how useful this poster would be unless you were trying to convince your colleagues to use the RUP.

I do not think that this book would be a sole point of reference if you were planning on conducting a development project using the RUP; however, it is a very clear introduction to the topic. It prompts thought in some areas of design, implementation and integration that may seem like common sense, but on which we would probably all benefit from some guidance.



Test Driven Development: A Practical Guide by David Astels (0 13 101649 0), Prentice Hall, 562pp @ £31-99 (1.25)
reviewed by Anthony Williams

This book is subtitled "A Practical Guide", and it is definitely that. It includes an overview of many of the tools available to assist you in using Test Driven Development (TDD), along with detailed descriptions of how TDD works in practice. The bulk of the book is taken up with working through a complete project from start to finish using TDD. This demonstrates many of the techniques in action, and gives examples of how to test different types of behaviour, including the GUI, which is notoriously hard to test. The book also includes appendices giving an overview of eXtreme Programming and Agile Modelling.

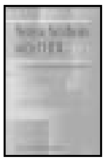
Most of the book uses Java as the target language, and it does appear that the majority of tools available are for Java. However, that does not mean that the techniques discussed are not applicable to other languages; there are chapters in this book dedicated to the use of the "xUnit" testing framework for various languages "x" – such as C++, Python, and Visual Basic.

The main thrust of TDD is that you write tests for the expected behaviour of the production code **before** you write the code, and that you don't add new code to the system **except** in response to a failing test. This takes some getting used to, but the key benefit is that you get a high-coverage automated test suite that automatically tells you if you break something when you make a modification.

As programmers, we all know that automated unit tests are a Good Thing, but writing tests after the fact is difficult, and seems like a chore since we already "know" the code works. TDD is not like that. I've been trying it on my latest project, and I am now "test infected" – I'd rather work this way than how I worked before. Applying the techniques discussed in the book to C++ was relatively straightforward; with the help of a few editor macros, adding a new test takes just two key presses, and I've managed to add the automated tests to the build script, so they appear as compile errors in the IDE.

I would recommend this book to all developers, especially those who find writing tests tedious or unnecessary, and those who wish they had a better set of tests when modifying code. Though TDD is one of the key techniques used in Agile methodologies, it can be used under any methodology, since it just replaces the developer tests that most developers would like to have anyway (but probably don't). Highly Recommended

VHDL



System Synthesis with VHDL by Petru Eles & Krzysztof Kuchcinski & Zebo Peng (0-79238-082-7), Kluwer Academic Publishers, 348pp @ £105-00 (1.60)
reviewed by Colin Paul Gloster

This book is aimed at advanced students reading design automation, though I think that really the main thrust of the book is better suited to people starting a career in writing VHDL compilers. It is a bit hard to see who this book is really best for as anyone who would read it would already be familiar with much of what it says and would only get an overview of new areas. The introductory chapters take up nearly half of the book and are intended to be skipped if the reader already knows about their topics. Reading the chapters sequentially is slightly repetitive because later chapters recapitulate a little of what is in introductory chapters.

Much of the book is fairly theoretical without being impenetrable, so can serve well as a review of the some 200 referenced papers.

One surprising thing to note about the authors' technique is that they claim that a user's behavioural model is converted very early on into a structural version but they use extra steps to reduce the required number of gates (thereby yielding a space-efficient but slower solution). A lot of the book is actually about restructuring graphs for scheduling constraints and to reduce the gate count so could really be read by someone with no interest in programmable logic devices ... the Kernighan-Lin algorithm; linear programming; simulated annealing and others are given varying degrees of coverage. Almost every problem discussed is NP-complete or NP.

One of their system level VHDL models is perhaps fairly small at 730 lines of code. Its code does not appear in the book. There is no substantial example of VHDL in the book: none is bigger than a page and most are less than half a page in length. The introductory chapter on VHDL is nice and covers more than enough of the language to understand what appears in the book. I found it convenient for contrasting VHDL93 (which they call VHDL'92) and VHDL87. VHDL is the input language for the authors' compiler, but they also survey system-level tools for other hardware description languages such as C^x and Verilog and HardwareC.

I learnt from the low-power synthesis chapter that two's complement is believed to consume more power than sign-magnitude due to the high level of switching needed if a variable/signal toggles between positive and negative often.

The final chapter often cites one reference that is missing from the bibliography.

Reading this book may be worthwhile, but you will not be able to work the way it advocates with ordinary tools so its impact on your work will be limited unless you decide to acquire an environment similar to those listed in the book. For some reason there is a chapter on testing which may encourage a design for testability culture ... but as with elsewhere in the book, it would be a good idea to check other texts for a practical way of implementing the notion.

VHDL Answers to Frequently Asked Questions 2ed by Ben Cohen (0-79238-115-7), kluwer Academic Publishers, 339pp @ £102-00 (1.60)
reviewed by Colin Paul Gloster

Six years on, this is still recommended. It explains many issues one might not have ever thought of. The chapters can be read in any order.

Most of the questions are of the genres "Why does the code shown in figure 2.5-1 yield an error?" or "How can a model be created to force errors onto a signal?". Cohen wrote many scenarios well.

Cohen explains that VHDL does not allow anonymous array types (which Ada gurus disapprove of anyway). Curly bracket huggers may have a little exposure to this kind of type: they are like Java's anonymous classes but unlike Java's anonymous arrays (sic) and unlike C++ anonymous unions. Anonymous non-array types are legal in VHDL and are not explicitly mentioned.

The earliest chapters fuel dissatisfaction with what is in other ways a nice language. Given solutions are not always ideal. Vendors are to blame, not Cohen.

The most important part of the book is its listing of VHDL features which most synthesizers refuse to produce a netlist for. Not an authoritative overview of modern tools, but it is bewildering that compilers for primitive programming languages had always given full support to features present in VHDL when synthesis tools do not support these features, and the lack of maturation in tools to support VHDL is shocking.

I am not keen on the favouring of weaktyping with subtyping in the book instead of strongtyping.

One section is intended as "a university type of final exam". Its layout is impractical for a dedicated student to pretend to be in an examination.

Much of two chapters are essentially pre-written by other people so you might not want to get the book. The index needs improvements.

Other material you might find without buying this book is in the news:comp.lang.vhdl FAQ. The book is better at giving solutions to less than perfect situations. Though old, the book does not lag behind. Neither the file nor the book devote much to the differences between the different versions of VHDL.

VHDL: Analysis and Modelling of Digital Systems by Z Navabi (0-07-046472-3), McGraw-Hill, 315pp @ £64-95 (?)
reviewed by Colin Paul Gloster

This is a VHDL 87 book aimed at students. The teaching of VHDL does not start until some thirty pages into the book, after tiny examples in lesser-known languages such as the Genrad Hardware Description Language (GHDL, which is not to be confused with Tristan Gingold's recent VHDL 87 simulator also named GHDL). No mention of Verilog is made at all.

The difference between inertial and transport delays is illustrated by a very clear diagram, whereas elsewhere in the book some other figures are not unclear but are worthless as the words are clear enough. Diagrams are used throughout the book but I, at least, found them to be typically overkill. I do not suggest that they be dropped for a future edition as they do not interfere with reading the chapters if you want to ignore them and people with different backgrounds may rely more heavily on different chapters' illustrations.

Even for people without any prior programming or coding experience, I doubt it is necessary to continue drawing diagrams breaking up code into statements and other syntactic parts as late as the second last chapter; or to follow a very well laid out code snippet of an assignment of all elements of a two dimensional array with a table showing the contents of the array. Some chapters are accompanied by traditional circuit symbols to complement the source code. This is fine for the very small examples novices are shown at first, but fortunately most of the more complicated parts of the book do not have these diagrams. Since the book was published, software 'engineers' have widely adopted drawing aids such as UML tools with relatively poor feasibilities for mapping to runnable executables. I do not expect Zainalabedin Navabi to have commented on this in his 1993 hardware book, but he should have advertised how preferable the flexibility of text-based hardware description languages are to once-popular schematic-based CAD tools.

An important difference in practice between inertial and transport delays is not mentioned even once: i.e. that synthesizers do not allow transport. Transport delays were really only in the book because they are in the language, and are downplayed in favour of inertial delays. Unfortunately, the author encourages many unsynthesizable constructs. Not only are Navabi's behavioral examples unsynthesizable (as is often the case with many people's simulation code), most of his RTL (register transfer level) code is too unsynthesizable and he misleads how low-level parts of VHDL are by claiming "guarded assignments, and resolution functions, which are considered to be among the most important hardware related constructs in the VHDL language".

Guards are used against Ben Cohen's advice. Zainalabedin Navabi extensively uses 'event outside IF and WAIT statements, this synthesis no-no goes as ever without any warning in this book aimed at learning hardware engineers.

Navabi does not give the low-down on how buggy some VHDL analysers can be; how simulators let a design perform feats which would be rejected by a synthesizer; how FPGAs intended for long-life products become ignored when vendors are only interested in supporting newer models; how very complicated it can be to determine from a manufacturer's datasheets how fast a FPGA is etc. A book could not really stay up-to-date with detailed lists of this nature, but a student needs to learn about harsh practicalities at some time. Michael John Sebastian Smith in his "Application-Specific Integrated Circuits" makes a good effort in imparting appreciation of these kinds of issues. Smith's beginners' book has a single VHDL chapter that is 79 pages long. Smith assumes no VHDL knowledge, but it is not really possible to learn VHDL from his book. In contrast, Navabi's VHDL book has a generally good ordering of the presented language features that build upon gradually introduced pre-requisites.

One strength of Smith's book (or many other books) over Navabi's TWO paragraphs on sensitivity lists is that sensitivity lists are not trivialized.

Navabi made a few small typos, some of which occur in the source code. I don't think that any of the earliest pieces of code had any lexical mistakes. One of the first array assignments has an identifier misspelt but by this stage in the book a novice would be able to spot and rectify the error. VHDL's expressive advantage of array indexing with enumeration types is clearly explained and quickly covered. Navabi chose to illustrate this with a two dimensional array whose elements are also of the same type as the index types. This is not as contrived as it might seem, because the array is a logic table for a gate applicable to multi-valued logic. However, it may have helped a newcomer to see this preceded by a plainer example with unrelated types for the indices and elements.

Earlier in the book the concept of ideal bits is matured slightly into ternary and quaternary value logic more for the purpose of teaching about enumeration types than imparting an appreciation for the uncertainties or irrelevance of certain parts of hardware etc. It is a VHDL-87 book so it is fair enough that the Std_Logic_1164 package is not mentioned. Signals are discussed a lot, leaving a novice with little to ask. A full explanation of `when :=` is legal for a signal should have been in the book and I would have liked an example of a signal resolution function which does not simply loop through all the drivers.

More coverage of user-defined attributes would have been nice but more importantly Navabi gave no warning that even many predefined attributes are not supported by synthesis tools. I could try to complain about more such as a mistake in the BNF appendix I found by accident, but really as an introduction to the syntax of VHDL this is a nice book. Practical implementations of subsets of VHDL may be disappointing so the book should not have only warned that the most abstract architectures will not be synthesizable. A beginner to programming may be frustrated when confronted with a compiler error message, but if he or she scrutinizes the rejected attempt at C++ or Java code would agree that it is illegal. Unless armed with something similar to the table of supported and unsupported synthesis constructs in "VHDL Answers to Frequently Asked Questions" by Ben Cohen, a beginner to describing hardware may be even more unhappy with an analyzer which rejects textbook code which is legal VHDL

Management

Effective Project Management 3ed by Robert Wysocki et al. (0 471 43221 0), Wiley, 452Pp+CD @ £34-95 (1.43)
reviewed by Giovanni Asproni

This book is about general project management techniques. The target readership is mainly practicing project managers, and teachers of project management at a university or college level.

It comes with a CD containing a trial version of MS Project 2002 plus some bonus material.

The book is structured in three main parts: the first part, is about traditional project management; the second part is about adaptive and extreme project management-according to the authors, these are new management techniques influenced by the agile software development community; the final part is about some organizational considerations, in particular project portfolio management, that is about company level strategies for project prioritisation and funding to achieve predetermined investment objectives.

I think this book is not a good buy for the following reasons.

It is difficult to use for self-study: the explanations are superficial, and, even if there is a bibliography section, the references to existing literature in the rest of the book are rare.

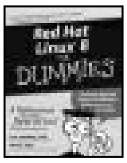
It is difficult to use as a reference: it is written to be read from cover to cover.

It is verbose: many concepts that could be easily explained with a simple picture, are introduced using wordy definitions-an example of that, is the definition of GANTT chart at page 119 (the first picture of it appears at page 220).

Finally, it has not enough examples, and the case study is not really helpful-it must be completely worked out by the reader, and neither the book nor CD have any hints about possible solutions to it.

Not recommended.

OS



Red Hat Linux 8 for Dummies by Jon Hall and Paul G.Sery (0-7645-1681-7), Wiley, 362c2pp @ £22-50 (1.33)
reviewed by Ian Bruntlett

Includes Publisher Edition of Red Hat Linux 2 CDs containing GNU compiler, Apache, Gnome & KDE. According to the back page this book needs a Pentium class PC with 32MB RAM CD drive and 650MB-2.5GB of hard disk capacity however according to p343 at least a 200MHz Pentium is required. *[This is incorrect. RedHat 8 requires 64Mb for its installation due to a bug in the installer which meant that the X server was started irrespective of whether a text installer was used or not - PFJ]*

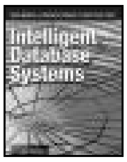
This book's blurb states "Install and start using Red Hat Linux today – no experience required" and "Includes Red Hat 8 Publisher's Edition on two CD ROMs". Well this is wrong in terms of experience required – this book is an impressive piece of work but any Unix/Linux neophyte armed with this book will need a lot of patience or will have a friend with system administration experience.

At the front of the book is a tear out reference card (covering vi, system administration, file permissions and some popular commands). It is quite useful but it lets itself down by missing key bits of information out.

If you have an old PC that can't boot from CDROM then this book is worthless. The money spent on buying this book would have been better off spent buying a full version of Linux, spending time looking through HOWTOs and MAN pages and buying a book like Running Linux which has more trouble shooting than this book has.

The full review takes up 1500 words and should be viewed on the ACCU website before making a decision to buy this book.

Databases



Intelligent Database Systems by Elisa Bertino, Barbara Catania & Gian Zarri (0 201 87736 8), Addison-Wesley, 451pp @ £29-95 (1.84)
reviewed by Thomas Padron-McCarthy

In intelligent database systems, database technology is combined with techniques developed in the field of artificial intelligence.

Database systems have always been good at managing very large amounts of data, and they contain important functionality such as error recovery, concurrency control and security. They do, however, suffer from limited modelling capabilities, compared to the mechanisms for knowledge representation in AI systems.

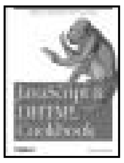
An intelligent database system combines the functionality expected from a database system with much better semantic support: new data models and new functionality that lets database designers specify what the data in the database actually mean. For example, instead of just organizing data in tables as in a relational database, rules expressed in a logic-like language can be used to deduce additional data, or to extract new knowledge about their relationship. Some of this exists in commercial systems, but mostly this is a research area.

Despite what the back cover claims, the coverage both of databases and of AI is much too brief to serve as an introduction to either field. If you want an introduction to databases, get yourself a database book. After that, you can read this book, if the area is of interest to you. If you are interested in AI in general, and not the more narrow part of AI that is relevant for intelligent database systems, get an AI book.

The book is, however, essential reading for a researcher or PhD student who wants to work in the field of intelligent database systems. It is also useful and interesting for someone who has some experience with databases, and who wants both a short introduction to some advanced database topics (such as temporal databases and mediators), and a more in-depth introduction to intelligent database systems.

It may be too specific and condensed to be very useful, or easy to read, for the ACCU member who is more of a general programmer and who has less database experience.

The Web & Networks



JavaScript and DHTML Cookbook by Danny Goodman (0-596-00467-2), O'Reilly, 520pp @ £28-50 (1.40)
reviewed by Alyn Scott

This is another of O'Reilly's Cookbook series covering JavaScript and Dynamic HTML. It provides code fragments and complete examples for web site designers to incorporate into their own web pages.

The presentation is quite clear with a discussion and detailed description of each 'recipe'. It can be somewhat dry reading unless you are able to try out the examples. A few more pictures would also have been helpful. All the sample code is available for free download on O'Reilly's web site.

There are 15 chapters in 522 pages that cover a broad range of subject areas. It starts with simple concepts like variables, arrays, functions, etc and then goes into more detailed examples. Some complete libraries of useful functions are provided, such as cross-browser positioning of HTML elements, creating custom scrollbars, etc. There is quite a large section on creating custom menus in various styles.

XML is all the rage these days and this book includes sections on importing XML data. The examples use XML-formatted data for custom menus and converting into JavaScript objects for loading into tables.

The discussions recommend good site design practices, such as not opening sub-windows and supporting all the common web browsers. Mac users are not excluded either.

I thought that this book would be very useful for anyone creating their own dynamic web sites. I did not have time to try out many of the examples, but the source code looks to be well written. It is well commented and described, so it should not be too hard to customise for your own personal requirements.

Apache: The Definitive Guide 3ed by Ben Laurie and Peter Laurie (0-596-00203-3), O'Reilly, 568pp @ £28-50 (1.40)
reviewed by Ian Bruntlett

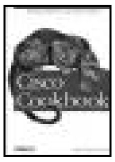
This book is for people who have used the web and have decided to run their own web servers and sites for themselves.

This book explains what a web server is and how it works. It explains web servers from Apache's point of view beginning with a simple web site and finishing off with a more capable web site running under Apache (1.3 or 2.0) under Windows or one of the Unixes (like Linux or BSD)

Chapter one introduces the reader to the basic concepts of a web site while chapter two gives the reader an introduction to running a simple web site, complete with an example. The authors then use the remaining twenty chapters to running industrial strength web sites. They cover server house keeping, virtual hosts, authentication, indexing, proxying, logging, security, big web sites, building applications, server side includes, PHP, CGI & Perl, mod_perl, mod_jserv, XML, the Apache API (latest v 2.0 and older v1.3) and how to write an Apache module.

Chapter 2 introduces the first tutorial web site, site.toddle. After some hard work, I got Apache to look at the first example site, site.toddle. Apache dealt with that success by reporting "out of memory" and plain quitting. Chapters 3 and 4 build upon the previous chapter's work. Both detail configuration settings.

This is a temporarily suspended review, which will be completed later on in the year, hopefully when I have access to better facilities.



Cisco Cookbook by Kevin Dooley and Ian J Brown (0-596-00367-6), O'Reilly, 885pp @ £38-95 (1.41)
reviewed by Alyn Scott

Cisco's IOS is huge. You just won't believe how mind-numbingly huge the IOS is. This book is full of proven 'recipes' for network engineers to use to solve problems configuring Cisco routers. I did not expect to read the book from cover to cover but it is full of techniques I had never thought possible. It should probably have "don't panic" written in large friendly letters on the cover.

This book will appeal to a limited number of people due to its rather specialised nature. You will need a networking qualification to understand the subject matter. Some of the examples are very trivial, while in others whole programs are provided, written in Perl or Expect languages, to streamline the configuration of large networks.

In 23 chapters and 870 pages it manages to cover a vast array of subjects, most of them related to TCP/IP. This includes most of the major routing protocols, plus router authentication, Frame Relay, tunnels and VPNs, dial backup, SNMP, access lists and multicast. There are many other aspects of router management that are comprehensively dealt with, but it's impossible to list them all here. This is certainly not light bedtime reading!

Each chapter has an introduction of between 2-5 pages that explains the subject quite concisely. Each 'recipe' is described in enough detail to understand the technique being demonstrated, but it helps if you have previous knowledge of the subject. The only major omission is a glossary for the hundreds of acronyms that this book introduces. However, the glossary might have been as big as the whole book!

Overall, the book fulfils its purpose very well. Networking is a huge subject and the authors have chosen a good range of topics to cover. If you are a network engineer this book will be a boon.

BGP by Iljitsch van Beijnum (0-596-00254-8), O'Reilly, 272pp @ £28-50 (1.40)
reviewed by Catriona O'Connell

The full title of this book "BGP: Building Reliable Networks with the Border Gateway Protocol" reflects the content of this book. It is about using BGP as a component in network design.

BGP is the major EGP (Exterior Gateway Protocol) in use today. It is used to route traffic between ISPs and those sites that are multihomed (connected to more than one ISP). The intended reader would probably work for an ISP or on a site that has connections to multiple ISPs.

This book takes the reader through the process of configuring a Cisco router from a non-BGP configuration to a working BGP configuration. If you have other routers, then this book will be of less value.

Throughout the book there are useful tips and pitfalls, highlighted by margin icons. It would be even better if the tips and traps could have been collected together in one table.

The book starts with the obligatory throwaway chapter on the history of the Internet, but chapter 2 introduces us to BGP, Multiprotocol BGP (MBGP), IPv6 as well as a few pages on common Interior Routing Protocols. Chapter 3, on physical network design is not specific to BGP. Chapter 4 introduces the concept of an Autonomous System (AS). ASs are fundamental to BGP as they are the objects between which routing takes place. Chapters 5 and 6 are about the use of BGP. Chapter 7 is on security. Again, only half of this chapter is BGP-related, the first half being a general discourse on security issues. Chapter 8 on running a NOC is superfluous. Chapter 9 on troubleshooting was disappointing. Half of it was related to the general process and only the second half looked at BGP-specific issues. This section would have been improved if network packet trace dumps had been shown. Chapters 9 through 12 are of primary relevance to large networks and ISPs.

I remain to be convinced of the value of the appendices. Anyone configuring BGP should have enough knowledge to configure a router and know enough about binary logic and netmasks. If they did not I would be very worried about letting them loose on a real network.

The glossary at the end of the book lists some common networking terms, but omits the plethora of 2, 3 and 4 letter acronyms that are associated with BGP.

As a readable primer on BGP, this is a good book. For more detailed information you would be better advised to go to the author's website (www.bgpexpert.com) or pick another book. Personally I would have liked to have seen the non-BGP material replaced by more detailed BGP material.

If you are a network engineer who needs to get up to speed quickly on implementing BGP on Cisco routers then I would recommend this book.

Practical VoIP Using Vocal by Luan Dang et al. (0 596 00078 2), O'Reilly, 502pp @ £31-95 (1.41)
reviewed by Silvia de Beer

VoIP (Voice over IP) is also known as packet telephony and enables you to make phone calls over the Internet. Initially this technology emerged to bypass expensive long-distance call charges. Since this technology has become more mature, VoIP is seen as a possibility to offer flexible new services to its users, without the slowness and the difficulty to change a Public Switched Telephone Network. VOCAL is open source software that enables a network to support VoIP. VOCAL is written by the start-up Vovida, now part of Cisco Systems. VOCAL is running on Linux, and everybody can set up a simple VoIP telephony network at home. The book describes version 1.3, but the latest release is already 1.5 at the time I was reading the book. The book describes how to install VOCAL, and perform your first tests and make simple phone calls. VOCAL is scaleable, and you can set up many services on numerous servers. You can do this without any coding, just by configuration. However if you want to offer new services, which are not provided in VOCAL, you would have to dive into the C++ code.

The authors do not assume that you have any telecom knowledge. All protocols, like SIP, RTP, MGCP, H.323, SNMP, which are used in VOCAL, are clearly explained in simple words and many message flow diagrams. Many chapters in this book describe the architecture and class structures of VOCAL, introducing you to the software (written in C++), in case you want to contribute to the open source project.

The screenshots of the provisioning server (responsible for maintaining and storing data and parameters needed by other servers) are described in too much detail; I assume that most people know that they need to click an OK button. The text is clearly written and is pleasant to read, there is no boasting about how perfect VOCAL is, and bugs or performance issues are honestly described.

BLAST by Ian Korf, Mark Yandell, & Joseph Bedell (0-596-00299-8), O'Reilly, 339pp @ £28-50 (1.40)
reviewed by Ivan Uemlianin

BLAST – the Basic Local Alignment Search Tool – is a set of programs for sequence alignment tasks in bioinformatics. The authors have produced a nicely written and thoughtful text. I cannot highly recommend it however, as the quality control is so poor.

The book suffers from very poor quality control: typos, broken cross-references, code or tables that do not tally with their text, inconsistent equation and citation formatting, unfinished glossary and index. It feels like a rough draft.

The book is in six parts. Part 1 introduces BLAST and walks through a search using BLAST's web interface [1]. Parts 5 & 6 include reference material: especially useful as the original programs do not have man pages.

Part 2 reviews molecular biology and approximate string matching. This is sketchy given space constraints, but it's not rushed, and it sticks to the point: to improve your understanding and use of BLAST.

Part 3 looks first at how BLAST uses the theory in part 2, and then puts this in context by discussing usage. This latter discussion is at a higher level to the similar discussion in the reference chapters.

Part 4 discusses installation, and some of the sysadmin necessary when running large analyses on large databases.

A recurrent theme is that BLAST searches should be treated as scientific experiments, and over half of part 3 takes this perspective. The BLAST programs have a vast array of command-line parameters and chapters 8 ('20 tips to improve your BLAST searches') and 9 ('BLAST protocols') discuss how to vary these parameters for common bioinformatic searches or experimental problems.

The discussion on system administration is equally high-level and intelligent, covering such things as local vs remote databases, optimising your hardware, and juggling your available CPU time.

The authors keep a tight rein on the subject matter, and some topics are cut off rather abruptly, especially in part 2, but this is not entirely inappropriate and references are given for further reading.

Notwithstanding its problems, this book is nicely written and well thought out. The writers have done a good job of cementing a fairly concrete collection of themes into a satisfying package.

I can recommend this book for practising bioinformaticians, and sysadmins of commercial or academic establishments. For these people the lack of quality control will be outweighed by the book's usefulness.

[1] `www.ncbi.nlm.nih.gov/BLAST`

Webmaster in a Nutshell 3ed by Stephen Spainhour and Robert Eckstein (0-596-00357-9), O'Reilly, 561pp @ £24-95 (1.40)
re-reviewed by Christopher Hill

Following a previous Highly Recommended review I would like to issue a word of warning.

This third edition is dated 2003; the second edition came out in 1999. To quote from the preface "What was relevant when we did the second edition in 1999 is still relevant today". Well I am not so sure. I have the impression that this is little more than a reprint, with a revised preface and few other minor changes, which makes this quite a dated book.

Chapter 3 the HTML reference talks of the "new" HTML 4.01 specification, and urges caution in using the advanced table elements (<THEAD> <TBODY> and <TFOOT> as "the latest browsers do not support them". This may have been true in 1999, but is no longer applicable in 2003. The chapter also does not indicate which elements are standard, and those that are vendor specific (i.e. <MARQUEE> and <MULTICOL> to pick two at random); nor is there any indication of deprecated elements. I also found there was poor cross-referencing between related elements; for example the <LI> element does not mention <UL> nor <OL> elements.

Chapter 9 covers CSS1; CSS2 became a W3C recommendation in 12 May 1998, but does not get a mention in this 2003 third edition. Interestingly the text claims to use Netscape and Internet Explorer icons to show which browser supports each property (as it did in the HTML section). Neither section has these icons in my printing.

So while this may be a useful single book to take on site to jog the memory, it will not inform a newcomer, nor satisfy the expert, nor touch on more recent specifications.

reviewed by Francis Glassborow

I went out of my way to get a copy of this book because I felt far too ignorant of the subject to understand how the various aspects of the website I maintain for my book worked. The publishers do not appear to be very keen to sell copies because when I dropped by Blackwells to get a copy there were none on the shelves. One of my friends in Blackwells searched their database and discovered that the book had been on order for eighteen months with a second order pending after six months. Eventually the ordered copies (well actually for the second edition though the original orders had been for the first) arrived almost a fortnight after a telephone call chasing them.

Was it worth the wait? Only because I was not paying for my copy.

To start with anyone who did not have some prior experience would not make head nor tail of the explanation of Cascading Style Sheets provided in the first few chapters of the book. These meander, use jargon and show no real grasp of what the newcomer might need.

The rest of the book is at least 50% too long because it is packed with unnecessary and often positively unhelpful repetition. For example the section on '*margin*' separately describes '*margin*', '*margin-left*', '*margin-right*', '*margin-top*' and '*margin-bottom*'. Of course with the tiniest of changes the sections are identical all the way down to which browsers support them.

What I really need is a single block description of the whole family coupled with notes about any ways in which individuals might be different.

I found it depressing to read through the lists of browsers that do not support a particular feature. Describing these as 'unsafe' seems an unnecessary euphemism for 'not implemented.' And I can see very little benefit in knowing that something is partially supported by a browser. I want my site to be as independent of browsers choice as it can be. In that light it would have been helpful had the author spent some time on how to bomb-proof HTML so that impoverished third world people using ancient browsers could still see something readable.

Yes, I will use this book for reference but I will not recommend it because it would be over-priced at any price in excess of $10 or £5. If the author is contemplating a third edition he should start to do a complete cover to cover rewrite. Start with a clear introduction to CSS, follow with a greatly condensed reference section and conclude with how to use CSS to make sites more portable across generations of browsers.