The magazine of the ACCU

www.accu.org

{cvu}

Volume 20 Issue 3 June 2008 £3

Features

Regulars

Pete Goodliffe
Write Less Code!

{cvu}
Volume 20 Issue 3
June 2008
ISSN 1354-3164
www.accu.org

ACCU vs. Open Source

I took on editorialship of C Vu around about the same time that I started working for Canonical. Canonical is an interesting company full of smart people. These are people who are leaders in their field and who care about what they do. Sound familiar? I though so. I thought that these people would be great ACCU people. They had similar ideals and to me everything seemed like a natural fit. Why then do I have such a hard time getting them to even consider ACCU?

I've come to realise over time that many of the things that people join ACCU for are also available through getting involved in open source projects. People get involved with ACCU to get around like minded people, who care about their profession. To be able to connect in some way with other people that they do not work with on a day to day basis that they can learn from, mentor, or just share a laugh with.

However all of these things are available through getting involved with an open source project that interests you. The projects are normally lead by a number of smart people. People who are interested in getting others interested in what they do. People who lead and mentor junior team members in how to fix problems, and follow the standards of the greater team. Members of these teams form bonds with others in the team. Often the members have never met, some occasionally try to get together and meet face to face. Now, the chances are if you have never been to an ACCU conference, I have no idea who you are, nor you me. This isn't so different to these open source projects, except that ACCU doesn't have an underlying project.

Now I don't know the actual numbers, but it is my guess that the vast majority of ACCU members are not actively involved in open source projects. It seems to be that ACCU offers the professional developer a way to get this camaradarie. Many upcoming developers though, those that are finishing degrees and entering the work force now have had some of the open source goodness. A good question then is 'What can ACCU offer over an above what they get from their open source participation?'

I feel like I'm running out of space, but one theme that has come up over the years of membership is of mentoring and apprenticeship. Perhaps there is something there? Perhaps you have other ideas. I, for one, would love to hear them.

TIM PENHEY,
EDITOR

The official magazine of ACCU

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

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

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

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

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

DIALOGUE

FEATURES

COPY DATES

C Vu 20.4: 1st July 2008
C Vu 20.5: 1st October 2008

IN OVERLOAD

Richard Harris continues to unravel knots in the second part of his series, Stuart Golodetz introduces the mathematics behind the RSA encryption algorithm, Tom Gilb presents a 'Quality Manifesto' and Klaus Marquardt describes the symptoms of 'Performitis'.

ADVERTISE WITH US

COPYRIGHTS AND TRADE MARKS

Garbage Collection in C and C++

Renato Forti provides automatic dynamic memory management for C and C++

Audience

This article was written for beginners and experienced developers who are not familiar with Garbage Collection and would like to have a small introduction.

Introduction

Traditionally in C, you need to dynamically allocate and deallocate memory using **`malloc`** and **`free`** or in C++ using the **`new`** and **`delete`** operators. This method is not ideal as managing dynamic memory by hand can be very hard and error prone.

Some languages, like Java, Smalltalk and Prolog, have a mechanism known as Garbage Collection (GC) that recycles memory automatically.

In this article I will introduce a fantastic GC library implemented by Hans J. Boehm[1], Alan Demers[2], and Mark Weiser[3] that brings automatic memory recycling to your C or C++ application.

The Boehm-Demers-Weiser garbage collector

The GC library is supported on various platforms, but not all, as GC cannot be implemented as completely portable C code. The current supported platforms include (at time of writing) Linux, Windows, MacOS X, and a variety of Unixes.

If your platform is not supported you could try to port yourself using Boehm's guidelines[4].

GC is open source. The licence is here: http://www.hpl.hp.com/personal/Hans_Boehm/gc/license.txt.

The GC library is well established and used in many projects around the world including Mozilla[5], the Irssi IRC client[6], the Berkeley Titanium project[7], and many others.

automatic memory recycling can significantly reduce the development time of your application

How GC works

The main function of GC is to provide an easy way to manage the memory of your program. You use **`new`** to allocate memory, but a call to **`delete`** is optional. If you don't call **`delete`**, the GC will reclaim the unreachable memory and make it available to your program again automatically.

Automatic memory recycling can significantly reduce the development time of your application, reducing errors caused by dynamic memory management that need the use of **`delete`** or **`free`**. Amongst other things, it can also be used as a leak detector. Memory reclaimed by the GC must not have been released by **`delete`** or **`free`**, and so must represent a leak. Mozilla, for example, uses GC to do this.

Boehm maintains a discussion about the arguments for and against garbage collection on his website[8], and I don't intend to repeat them here.

GC uses the mark-sweep algorithm. This algorithm determines what memory can be recycled by occasionally marking all objects referenced directly by pointer variables. It marks all objects directly reachable from newly marked objects, then a sweep over the entire heap is performed to restore unmarked objects to a free list. These objects can then be reallocated. You can find a complete discussion of the mark-sweep algorithm on Boehm's website[9].

Get and build

When I wrote this article the most recent version of GC was 7.0. You can download it from:

http://www.hpl.hp.com/personal/Hans_Boehm/gc/gc_source/

After downloading, use the provided `MAKEFILE` to build the library in `gc-root/doc`. The readme files are very useful for learning how to use the `MAKEFILE` for your platform.

In my case I will build GC for Windows and so my readme is `README.win32`.

To build for Win32 we need rename: `NT_THREADS_MAKEFILE` (it is in: `gc-root/`) to `MAKEFILE`.

Now do the build. It will create a `gc.dll`, and link library `gc.lib`. If you need a static library you can use `NT_STATIC_THREADS_MAKEFILE` to generate a `.lib` instead. On Windows you also have have other options:

- `NT_MAKEFILE`
- `NT_STATIC_THREADS_MAKEFILE`
- `NT_THREADS_MAKEFILE`
- `NT_X64_STATIC_THREADS_MAKEFILE`

See readme for details on your platform.

Binary packages may also be available through the usual channels. The GC library is, for example, included in Debian (**`apt-get install libgc-dev`**) and is available for MacOS X through MacPorts.

C++ interface

Most of the GC library is written in C. The C++ interface is implemented as a layer on top of the C interface. We will use this interface here. If you want use the C interface see the online guide[10].

The C++ interface is defined in `gc_cpp.h`, and you can find this file in `gc-root/include/gc_cpp.h` and the implementation in `gc-root/`. All clients should include this file.

Listing 1 is the GC C++ interface (`gc_cpp.h`).

As you can see it provides new, new[], delete and delete[] operators for a class. It also provides replacements for the global new and delete operators. This can be a problem on some platforms and you may need do some adjustments in the C++ layer (gc_cpp.cc, gc_cpp.h, and possibly gc_allocator.h) for correct operation.

Using GC.

We will start with a simple example that makes a class collectable to GC. Listing 2 shows an ordinary class. When complied and run the output, shown in Listing 3, is as you would expect.

RENATO FORTI

Renato Tegon Forti is an independent software developer, working from his home office in Brazil for many companies. He can be contacted at re.tf@acm.org

Listing 1

```
class gc
{
public:
  void* operator new( size_t size );
  void* operator new( size_t size,
                      GCPlacement gcp );
  void* operator new( size_t size,
                      void *p );
  void operator delete( void* obj );
  void* operator new[]( size_t size );
  void* operator new[]( size_t size,
                        GCPlacement gcp );
  void* operator new[]( size_t size,
                        void *p );
  void operator delete[]( void* obj );
};

void* operator new[]( size_t size );
void operator delete[]( void* obj );
void* operator new( size_t size );
void operator delete( void* obj );

// many implementation details and preprocessor
// directives were omitted for clarity
```

Listing 2

```
#include <iostream>
class Test
{
public:
  Test()
  {
    std::cout << " Test Constructor..."
              << std::endl;
  }
  ~Test()
  {
  std::cout << " Test Destructor..."
            << std::endl;
  }
};
void test()
{
  for (size_t i = 0; i != 5; ++i)
  {
    Test* pTest = new Test();
    ...
    delete pTest;
  }
}
int main()
{
  test();
  return 0;
}
```

Listing 3

```
Test Constructor...
Test Destructor...
Test Constructor...
Test Destructor...
Test Constructor...
Test Destructor...
Test Constructor...
Test Destructor...
Test Constructor...
Test Destructor...
```

Listing 4

```
#include <iostream>
#include <gc/gc_cpp.h>
class Test: public gc // NOTE: inherit from gc
{
  //… as Listing 2 …
};
void test()
{
  for (size_t i = 0; i != 5; ++i)
  {
    Test* pTest = new Test();
    ...
    // delete pTest; // no longer needed
  }
}
int main()
{
  GC_INIT();
  test();
  return 0;
}
```

Listing 5

```
Test Constructor...
Test Constructor...
Test Constructor...
Test Constructor...
Test Constructor...
```

Now we will make the class collectable. This is very straightforward, as shown in Listing 4, we simply make our class publically derived from the **gc** base class.

The output from the modified program is shown in listing 5. Well, it shows 5 constructors and but what about destructors?

This occurs because a class that is derived from **gc** will only have its memory freed. The destructor will not be called. We will see how the destructor can be called below.

Continuing, if you need to make your class collectable by GC, do you need to inherit from **gc**? The class you wish to collect may not be under your control, perhaps from a third party library. Happily, there are other options.

For example you can use the GC placement new to make object instances collectable, as shown in listing 6.

The output from this program, shown in listing 7, is the same as the previous example. However, the difference to not is in listing 4, the class

Listing 6

```
#include <gc/gc_cpp.h>
class Test // NOTE: no longer inherit from gc
{
  //… as Listing 2 …
};
void test()
{
  for (size_t i = 0; i != 10; ++i)
  {
    Test* pTest = new(GC) Test(); // note (GC)
    ...
    // delete pTest; // no longer needed
  }
}
int main()
{
  GC_INIT();
  test();
  return 0;
}
```

Listing 7

```
Test Constructor...
Test Constructor...
Test Constructor...
Test Constructor...
Test Constructor...
```

is marked as collectable, while in listing 6, the individual object instances are marked as collectable.

Now if your class inherits from gc but you don't want GC to collect particular instances, you can use the NoGC placement new. Listing 8 shows this in action. Note that since we have turned off garbage collection for these objects we must explicitly delete them and therefore their destructors are called, as shown in listing 9.

Another common question is if my class is collectable and I use delete to free memory, what will happen?

The memory will be freed immediately, as expected and the destructor will run. You can explicitly delete both uncollectable and collectable objects.

gc_cleanup / call destructor

GC C++ interface provides a class called gc_cleanup. If your class inherits from it then it will be visible to the GC to be collected, and when an object is freed by the GC its destructor will be called. List 10 shows a class which inherits from gc_cleanup, and listing 11 shows the output from this program.

Clean-up function

The GC interface also provides a clean-up function that any collectable object may have. This function is invoked when the collector discovers the object to be inaccessible, like the **gc_cleanup** class.

Listing8

```
#include <iostream>
#include <gc/gc_cpp.h>
class Test: public gc // NOTE: inherit from gc
{
  //… as Listing 2 …
};
void test()
{
  for (size_t i = 0; i != 5; ++i)
  {
    Test* pTest = new(NoGC) Test();
    …
    delete pTest; // now we need to use delete
  }
}
int main()
{
  GC_INIT();
  test();
  return 0;
}
```

Listing 9

```
Test Constructor...
Test Destructor...
Test Constructor...
Test Destructor...
Test Constructor...
Test Destructor...
Test Constructor...
Test Destructor...
Test Constructor...
Test Destructor...
```

Listing 10

```
#include <iostream>
#include <gc/gc_cpp.h>
class Test :  public gc_cleanup
{
  … as Listing 2 …
};

void test()
{
  for (size_t i = 0; i != 5; ++i)
  {
    Test* p = new Test ();
  }
}

int main()
{
  GC_INIT();
  test();
  return 0;
}
```

Listing 11

```
Test Constructor...
Test Constructor...
Test Constructor...
Test Constructor...
Test Constructor...
Test Destructor...
Test Destructor...
Test Destructor...
Test Destructor...
Test Destructor...
```

GC determines if an object is inaccessible when you allocate memory and it becomes impossible to ever free it because there are no longer any references to it, then the clean-up function is invoked.

Listing 12 show an example, and listing 13 its output.

Listing 12

```
class Test : public gc
{
  public:
  … as Listing 2 …
  static void clean(void* obj, void* data )
  {
    std::cout << " clean function" << std::endl;
  }
};

void test(size_t sz)
{
  Test * p;
  for (size_t i = 0; i < sz; i++)
  {
    p = ::new (GC, Test::clean) Test ();
  }
}

int main()
{
  GC_INIT();
  test();
  return 0;
}
```

Write Less Code!

Pete Goodliffe implores us to produce less code for the sake of our software.

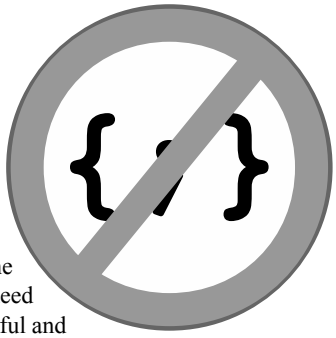
It's a sad fact that in our modern world that there's just too much code. I can cope with the fact that my car engine is controlled by a computer, there's obviously software cooking the food in my microwave, and it wouldn't surprise me if my genetically modified cucumbers had an embedded micro controller in them. That's all fine; its not what I'm obsessing about. I'm worried about all the *unnecessary* code out there.

There's simply too much unnecessary code kicking around. Like weeds, these evil lines of code clog up our precious bytes of storage, obfuscate our revision control histories, stubbornly get in the way of our development, and use up precious code space, choking the good code around them.

Why is there so much unnecessary code? Perhaps it's due to genetic flaws. Some people like the sound of their own voice. You've met them; you just can't shut them up. They're the kind of people you don't want to get stuck with at parties. Yada yada yada. Other people like their own code too much. They like it so much they write reams of it. `{yada->yada.yada();}` Or perhaps they're the programmers with misguided managers who judge progress by how many thousands of lines of code have been written a day.

Writing lots of code does *not* mean that you've written lots of software. Indeed, some code can actually negatively affect the amount of software you have – it gets in the way, causes faults, and reduces the quality of the user experience. The programming equivalent of anti-matter.

Some of my best software improvement work has been by removing code. I fondly remember one time when I lopped literally thousands of lines of code out of a sprawling system, and replaced it with a mere ten lines of code. What a wonderfully smug feeling of satisfaction. I suggest you try it some time.



Why should we care?

So why is this phenomenon bad, rather than merely annoying? There are many reasons why unnecessary code is the root of all evil. Here are a few headlines:

- Writing a fresh line of code is the birth of a little lifeform. It will need to be lovingly nurtured into a useful and profitable member of software society. Then you release the product.

  Over the life of the software system, that line of code needs maintenance. Each line of code costs a little. The more you write, the higher the cost. The longer they live, the higher the cost. Clearly, unnecessary code needs to meet a timely demise before it bankrupts us.

- More code means there is more to read – it makes our programs harder to comprehend. Unnecessary code can mask the purpose of a function, or hide small but important differences in otherwise similar code.

PETE GOODLIFFE

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

Garbage Collection (continued)

Conclusion

GC is powerful tool and can simplify the complexity of writing software. It can bring an easier way to manage memory like you have in Java, but for C++ users. The Boehm GC provides collection on a per class or per instance basis, with the option of calling or not calling destructors. The negative point is that it introduces a reduction of performance in system. This is not so much of a problem for most modern system. GC deserves a look. ■

Acknowledgements

I would like to thank Jez Higgins and Paul Grenyer for the various improvements they suggested for this article.

Listing 13

```
Test Constructor...
Test Constructor...
Test Constructor...
Test Constructor...
Test Constructor...
clean function
clean function
clean function
clean function
clean function
```

Bibliography

Richard Jones & Rafael Lins, Garbage Collection - Algorithms for Automatic Dynamic Memory Management, ISBN 978-0471941484

A garbage collector for C and C++ (Hans J. Boehm), http://www.hpl.hp.com/personal/Hans_Boehm/gc/

Libgc, http://developers.sun.com/solaris/articles/libgc.html

The Web Links listed here may not be valid in the future.

References

[1] Hans J. Boehm, http://www.hpl.hp.com/personal/Hans_Boehm

[2] Alan Demers, http://www.cs.cornell.edu/annual_report/00-01.bios.html#demers

[3] Mark Weiser, http://www-sul.stanford.edu/weiser/

[4] Porting guidelines, http://www.hpl.hp.com/personal/Hans_Boehm/gc/porting.html

[5] Mozilla project, http://www.mozilla.org/

[6] Irssi IRC client, http://www.irssi.org

[7] Berkeley Titanium project, http://titanium.cs.berkelyh.edu/

[8] http://www.hpl.hp.com/personal/Hans_Boehm/gc/issues.html

[9] http://www.hpl.hp.com/personal/Hans_Boehm/gc/complexity.html

[10] http://www.hpl.hp.com/personal/Hans_Boehm/gc/gcinterface.html

- The more code there is, the more work required to make modifications – the program is harder to modify.
- Code harbours bugs. There more code you have, the more places there are for bugs to hide.

Flappy logic is the sign of a flappy mind or, at least, of a poor understanding of logic constructs

- Duplicated code is particularly pernicious; you can fix a bug in one copy of the code and, unbeknown to you, still have another thirty two identical little bugs kicking around elsewhere.

Unnecessary code comes in many guises: unused components, dead code, pointless comments, unnecessary verbosity, and so on. Let's look at some of these in detail.

Flappy logic

A simple and common class of pointless code is the unnecessary use of conditional statements and tautological logic constructs. Flappy logic is the sign of a flappy mind or, at least, of a poor understanding of logic constructs. For example:

```
if (expression)
  return true;
else
  return false;
```

can more simply, and directly be written:

```
return expression;
```

This is not only more compact, it is easier to read, and therefore easier understand. It looks more like an English sentence, which greatly aids human readers. And do you know what? The compiler doesn't mind one bit.

Similarly, the verbose expression:

```
if (something == true)
{
  // ...
}
```

would read much better as:

```
if (something)
```

Now, these examples are clearly simplistic. In the wild we see much more elaborate constructs created; never underestimate the ability of a programmer to complicate the simple. Real World code is riddled with things like this:

```
bool should_we_pick_bananas()
{
  if (gorilla_is_hungry())
  {
    if (bananas_are_ripe())
    {
      return true;
    }
    else
    {
      return false;
    }
  }
  else
  {
    return false;
  }
}
```

which reduces neatly to the one-liner:

```
return gorilla_is_hungry() && bananas_are_ripe();
```

Cut through the waffle and say things clearly, but succinctly. Don't feel ashamed to know how your language works. It's not dirty, and you won't grow hairy palms. Knowing, and exploiting, the order in which expressions are evaluated saves a lot of unnecessary logic in conditional expressions.

```
if ( a
  || (!a && b) )
{
  // what a complicated expression!
}
```

can simply be written:

```
if (a || b)
{
  // isn't that better?
  // didn't hurt, did it?
}
```

Short-circuit evaluation is your friend.

Duplication

Code duplication is evil. We mostly see this crime perpetrated through the application of cut-and-paste programming; when a lazy programmer choses not to factor repeated code sections into a common function, but physically copies it from one place to another in their editor. Sloppy. The sin is compounded when the code is pasted with minor changes.

When you duplicate code, you hide the repeated structure, and you copy all of the bugs that existed. Even if you repair one instance of the code, there will be a queue of identical bugs ready to bite you another day. Refactor duplicated code sections into a single function. If there are similar code sections with slight differences, capture the differences in one function with a configuration parameter.

Not all duplication is malicious or the fault of lazy programmers. Duplication can happen by accident too, by someone reinventing a wheel that they didn't know existed. Or it can happen by constructing a new function when a perfectly acceptable third party library already exists. This is bad because the existent library is far more likely to be correct and debugged already. Using common libraries saves you effort, and shields you from a world of potential faults.

Not all duplication is malicious or the fault of lazy programmers

There are also micro code-level duplication patterns. If example:

```
if (foo) do_something();
if (foo) do_something_else()
if (foo) do_more();
```

could all be neatly wrapped in a single `if` statement. Multiple loops can usually be reduced to a single loop. For example, the following code:

```
for (int a = 0; a < MAX; ++a)
{
  // do something
}
// make hot buttered toast
for (int a = 0; a < MAX; ++a)
{
  // do something else
}
```

boils down to:

```
for (int a = 0; a < MAX; ++a)
{
  // do something
  // do something else
}
// make hot buttered toast
```

Not only is this simpler to read and understand, it's likely to perform better too, as only one loop needs to be run. Also consider redundant duplicated conditionals:

```
if (foo)
{
  if (foo && some_other_reason)
  {
    // the 2nd check for foo was redundant
  }
}
```

You probably wouldn't write that on purpose, but after a bit of maintenance work a lot of code ends up with sloppy structure like that. If you spot duplication, remove it.

I was recently trying to debug a device driver that was structured with two main processing loops. Upon inspection, these loops were almost entirely identical, with some minor differences for the type of data they were processing. This fact was not immediately obvious because each loop was 300 lines (of very dense C code) long! It was tortuous and hard to follow. Each loop had seen a different set of bugfixes, and consequently the code was flakey and unpredictable. A little effort to factor the two loops into a single version halved the problem space immediately; I could then concentrate on one place to find and fix faults.

Dead code

If you don't maintain it, your code can rot. And it can also die. *Dead code* is code that is never run, that can never be reached. That has no life. Tell your code to get a life, or get lost.

These examples both contain dead code sections that aren't immediately obvious if you were to quickly glance over them:

```
if (size == 0)
{
  // ... twenty lines of malarkey ...
  for (int n = 0; n < size; ++n)
  {
    // this code will never run
  }
  // ... twenty more lines of shenanigans ...
}
```

and

```
void loop(char *str)
{
  size_t length = strlen(str);
  if (length == 0) return;
  for (size_t n = 0; n < length; n++)
  {
    if (str[n] == '\0')
    {
      // this code will never run
    }
  }
  if (length) return;
  // neither will this code
}
```

Other manifestations of dead code include:

- Functions that are never called.
- Variables read but never written
- Parameters passed but never used
- Enums, structs, classes, or interfaces that are never used

Comments

Sadly, the world is riddled with awful code comments. You can't turn around in an editor without tripping over a few of them. It doesn't help that many corporate coding standards are a pile of rot, mandating the inclusion millions of brain dead comments.

Good code does not need reams of comments to explain it. Careful choice of variable, function and class names and good structure should make your code entirely clear. Duplicating all of that information in a set of comments is unnecessary redundancy. And like any other form of duplication, it is also dangerous; it's far too easy to change one without changing the other.

Stupid, redundant comments range from the classic example of byte wastage:

```
++i; // increment i
```

to more subtle examples, where an algorithm is described just above it in the code:

```
// loop over all items, and add them up
int total = 0;
for (int n = 0; n < MAX; n++)
{
  total += items[n];
}
// (yes, we could see that, thanks!)
```

Don't write comments describing what the code used to do; it doesn't matter any more

Very few algorithms when expressed in code are complex enough to justify that level of exposition. (But some are – learn the difference!)

It's also common to enter a crufty codebase and see 'old' code that has been surgically removed, by commenting it out. Don't do this; it's the sign of someone who wasn't brave enough to perform the surgical extraction completely, or who didn't really understand what they were doing and thought that they might have to graft the code back in later. Remove code completely. You can always get it back afterwards from your source control system.

Don't write comments describing what the code used to do; it doesn't matter any more. Don't put comments at the end of code blocks or scopes; the code structure makes that clear. And don't write gratuitous ASCII art.

Verbosity

This is really a quite general topic. A lot of code is needlessly chatty. At the simplest end of the verbosity spectrum (which ranges from infra-redundant to ultra-voluble) is code like this:

```
bool is_valid(const char *str)
{
  if (str)
    return strcmp(str, "VALID") == 0;
  else
    return false;
}
```

It is quite wordy, and so it's relatively hard to see what the intent is. It can easily be rewritten:

```
bool is_valid(const char *str)
{
  return str && strcmp(str, "VALID") == 0;
}
```

Don't be afraid of the ternary operator, it really helps reduce code clutter. Replace this kind of monstrosity:

```
public String getPath(URL url) {
  if (url == null) {
    return null;
  }
  else {
    return url.getPath();
  }
}
```

with:

```
public String getPath(URL url) {
  return url == null ? null : url.getPath();
}
```

Of course, you can't do that in Python – it doesn't have a ternary operator. Snakes can't count to three.

C-style declarations (where all variables are declared at the top of a block, and used much, much later on) are now officially passé (unless you're still forced to use officially defunct compiler technology). The world has moved on, and so should your code.

```
int a;
// ... twenty lines of C code ...
a = foo();
// what type was an "a" again?
```

Move variable declarations and definitions together, to reduce the effort required to understand the code, and reduce potential errors from uninitialised variables. In fact, sometimes these variables are pointless anyway:

```
bool a;
int b;
a = condition_1;
b = condition_2;
if (a)
 foo(10, b);
else
 foo(5, b);
```

can easily become the less verbose (and, arguably clearer):

```
foo(condition_1 ? 10 : 5, condition2);
```

Bad design

Of course, unnecessary code is not just the product of low-level code mistakes or bad maintenance. It can be caused by higher-level design flaws. For example:

- Bad design may introduce many unnecessary communication paths between components – lots of extra code for data marshalling for no apparent reason. The further data flows, the more likely it is to get corrupted en-route.
- Over time code components become redundant, or can mutate from their original use to something quite different, leaving large sections of unused code. When this happens, don't be afraid to clear away all of the dead wood. Replace the old component with a simpler one that does all that is required.

Whitespace

Don't panic! I'm not going to attack whitespace (that is, spaces, tabs, and newlines). Whitespace is a good thing – do not be afraid to use it. Like a well placed pause when reciting a poem, sensible use of whitespace helps to frame our code.

Use of whitespace is not usually misleading or unnecessary. But you can have too much of a good thing, and twenty newlines between functions probably is too much.

Consider, too, the use of parenthesis to group logic constructs. Sometimes brackets help to clarify the logic even not necessary to defeat operator precedence. Sometimes they are unnecessary and get in the way.

So what do we do?

To be fair, often such a build up of code cruft isn't intentional. Few people set out to write deliberately laborious, duplicated, pointless code. (But there are some lazy programmers who continually take the low-road rather than invest extra time to write great code.) Most frequently we end up with these code problems as the legacy of code that has been maintained, extended, worked with, and debugged by many people over a large period of time.

So what do we do about it? We must take responsibility. Don't write unnecessary code, and when you work on 'legacy' code watch out for the warning signs. It's time to get militant. Reclaim our whitespace. Reduce the clutter. Spring clean. Redress the balance. Pigs live in their own filth. Programmers needn't. Clean up behind yourself. As you work on a piece of code, remove all of that unnecessary code you encounter.

But take heed of this simple rule: make 'tidying up' changes separately from other functional changes. This will ensure that its clear in your source control system what's happened. Gratuitous structural change mixed in with functional modifications are hard to follow. And if there is a bug then it's harder to work out whether it was due to your new functionality, or because of the structural improvement.

Conclusion

Software functionality does not correlate to the number of lines of code, or to the number of components in a system. More lines of code does not necessarily mean more software.

So if you don't need it: don't write it. Write less code, and find something more fun to do instead. ■

Custom Iterators in C++

Jez Higgins searches for a base class.

Custom iterators in Java are easy: you just implement the `java.util.Iterator` interface. Listing One shows an example. The `FilterIterator` wraps an existing iterator, returning only those values which satisfy some condition. Doing the same thing in C++ is more awkward.

Listing 1

```
public interface Predicate
{
  public boolean test(Object o);
}
public class FilterIterator implements Iterator
{
  public FilterIterator(Iterator iterator,
                        Predicate predicate)
  {
    iter_ = iterator;
    pred_ = predicate;
    findNext();
  }
  public boolean hasNext()
  {
    return next_ != null;
  }
  public Object next()
  {
    Object current = next_;
    findNext();
    return current;
  }
  public void findNext()
  {
    next_ = null;
    while(iter_.hasNext() && next_ == null)
    {
      Object candidate = iter_.next();
      if(pred_.test(candidate))
        next_ = candidate;
    }
  }
  ...
}
```

Custom iterators and their uses, which include abstracting data sources, building views and queries, and pipeline processing, were the basis of a session I gave at the 2007 conference[1]. It had gone well, but I wasn't quite sure I had really put my main point across very well. In December last year, I had another go, presented a revised version to the ACCU Cambridge group[2]. The example code I show is in Java and Python, primarily for my own convenience. As I've noted, iterators in Java are easy and they are equally easy in Python, so you can fit a complete example on one slide. Further, most people can read Java and Python even if they don't use them.

The audience that evening was largely composed of guys who worked in C++. I asserted throughout that the ideas shown could be implemented in C++ without any particular difficulty. I was picked up on that by someone (whose name I unfortunately didn't catch, apologies) who argued that a C++ version of the `FilterIterator` would be hampered by the type signatures. They would simply get in the way. Java iterators can be freely interchanged at runtime, thanks to that base interface. C++ iterators have no such common base, and iterators are typically tied to the type of whatever they iterate over. A `std::vector<string>`'s iterators are of type `std::vector<string>::iterator`, a `std::deque<string>`'s iterators are of type `std::deque<string>::iterator`, and the two are not interchangable at runtime, despite both having essentially identical characteristics. Generally this isn't a problem but for the situations I was describing, which rely on runtime composition and substitution, it presents rather an obstacle. I continued to assert that the obstacle was, in fact, surmountable and that a little bit of scaffolding would take you a long way. He disagreed. There was a bit of back and forth around the room, but I didn't win him over and, short of writing code on the whiteboard, it didn't look like I was going to. So we all went to the pub.

In the following few days I did write the code, and I think I would have brought him round in the end.

JEZ HIGGINS

Jez works in his attic, which was recently replastered, living the devil-may-care life of a journeyman programmer. He is currently learning to tumble turn without getting water up his nose. His website is http://www.jezuk.co.uk/

A naive C++ filtering iterator

Listing Two shows my first pass at a C++ filtering iterator. There are some obvious differences from the Java version, primarily due to differences in language idiom. Java iterators know when they hit the end of the range they traverse, while C++ uses a pair of iterators to denote a range.

This iterator does indeed filter and its use, shown in Listing Three is straightforward. Job done, right? Well, no.

It works as a filter, but the usage is cumbersome at best. The type of the instantiated `filter_iterator` is related not only to the type of iterator it wraps but also to the type of the predicate. You cannot substitute a `filter_iterator<std::vector<int>::iterator, Even>` for a `std::vector<int>::iterator`, nor can you substitute it for `filter_iterator<std::vector<int>::iterator, Odd>`. I'm not sure that fits any accepted definition of adding flexibility. It certainly isn't any closer to the runtime behaviour I want.

But what to do?

Simplification through type erasure

The approaches I described in the talk are built on object-oriented techniques. The Java standard library is resolutely object-oriented, with concepts expressed as interfaces or classes. Huge chunks of the C++ Standard Library, on the other hand, are built around generic programming techniques, most notably the containers, iterators, and algorithms of the STL. Concepts are expressed as a set of requirements on a type, rather than directly in code. A Java iterator is an iterator because it implements the Iterator interface. A C++ iterator is an iterator because it can be can be dereferenced to refer to some object, and incremented to obtain the next object in a sequence[3]. Java iterators are substitutable at runtime, while C++ iterators are substitutable at compile time.

The proliferation of types in the code above arise from the tension between generic and object-oriented approaches. How can that tension be relieved?

In our `filter_iterator`, we don't care about the actual type of iterator being wrapped. It isn't exposed through the public interface, and so our client code doesn't (indeed can't) care about it either. In the public interface, we only care that it returns an `int` (or whatever) when

Listing 2

```
template<typename iterator_type,
         typename predicate_type>
class filter_iterator
{
public:
  typedef typename iterator_type
     ::value_type value_type;
  filter_iterator(const iterator_type& begin,
     const iterator_type& end,
     const predicate_type& pred):
current_(begin),
     end_(end), pred_(pred)
  {
    while((current_ != end_)
       && (!pred_(*current_)))
      ++current_;
  } // filter_iterator
  value_type& operator*() const { return
*current_; }
  value_type& operator->() const {
     return *current_; }
  filter_iterator& operator++() { advance();
     return *this; }
  bool operator==(const filter_iterator& rhs)
     const { return current_ == rhs.current_; }
  bool operator!=(const filter_iterator& rhs)
     const { return !(operator==(rhs)); }
private:
  void advance()
  {
    do
    {
      ++current_;
    }
    while((current_ != end_)
       && (!pred_(*current_)));
  } // advance
  iterator_type current_;
  iterator_type end_;
  predicate_type pred_;
};
```

dereferenced. Internally, the implementation only requires that wrapped iterator can be advanced by calling **operator++** and compared for equality. If only there was a common base class along the lines of Listing 4.

If you squint a bit, looks almost like the **java.util.Iterator**, doesn't it? Perhaps Josh Bloch was on to something after all. Anyway ...

Listing 3

```
class Even
{
public:
  bool operator()(int& i) { return i%2 == 0; }
};
...
std::vector<int> vec;
... populate the vector ...
filter_iterator<std::vector<int>::iterator,
   Even> fb(vec.begin(), vec.end(), Even());
filter_iterator<std::vector<int>::iterator,
   Even> fe(vec.end(), vec.end(), Even());
for( ; fb != fe; ++fb)
{
  std::cout << *fb << std::endl;
  ... do something else with *fb ...
} // for ...
```

Listing 4

```
class iterator_base
{
  virtual int& get(); // aka operator*()
  virtual void advance(); // aka operator++()
  virtual bool equal(
     const iterator_base& rhs) const;
  // aka operator==()
}
```

We can effectively introduce such a base class through the use of an adaptor or wrapper. If we have a template class, parameterised on the iterator's type, with a non-template base class, we can create a type-specific wrapper to swaddle around the iterator, which we only manipulate through the base class. Because the base class isn't parameterised, it doesn't expose anything about the wrapped iterator. By the classical computer science technique of an extra layer of indirection, we can slip a common base class in down the side of our existing iterators.

The adaptor is shown in Listing 5, which is continued on the next page.

Listing 5

```
template<typename value_type>
class iterator_holder
{
public:
  template<typename iterator_type>
  iterator_holder(const iterator_type& iter) :
      iter_(new holder<iterator_type>(iter)) { }
  iterator_holder(const iterator_holder& rhs) :
      iter_(rhs.iter_->clone(); }
  ~iterator_holder() { delete iter_; }
  value_type& get() const { return iter_->get();
}
  void advance() { return iter_->advance(); }
  iterator_holder& operator=(
     const iterator_holder& rhs)
  {
    iterator_holder(rhs).swap(*this);
    return *this;
  }
  template<typename other_iterator_type
  iterator_holder& operator=(
     constother_iterator_type& iter)
  {
    iterator_holder(rhs).swap(*this);
    return *this;
  }
  void swap(iterator_holder& rhs)
  {
    iterator_base* c = iter_;
    iter_ = rhs.iter_;
    rhs.iter_ = c;
  }
  bool operator==(const iterator_holder& rhs)
const
  {
    return iter_->equal(rhs.iter_);
  }
  bool operator!=(const iterator_holder& rhs)
const
  {
    return !(operator==(rhs));
  }

private:
  class iterator_base;
  iterator_base* iter_;
  class iterator_base
  {
```

Listing 5 (cont'd)

```
  public:
    virtual ~iterator_base() { }
    virtual iterator_base* clone() const = 0;
    virtual void advance() = 0;
    virtual value_type& get() const = 0;
    bool equal(iterator_base* rhs) const
    {
      return (type() == rhs->type())
         && (up_compare(rhs));
    }
  protected:
    virtual const std::type_info& type()
       const = 0;
    virtual bool up_compare(
       iterator_base* rhs) const = 0;
  }; // class iterator_base
  template<typename iterator_type>
  class holder : public iterator_base
  {
  public:
    holder(const iterator_type& iter) :
       iter_(iter)
    { }
    virtual iterator_base* clone() const {
       return new holder(iter_); }
    virtual void advance() { ++iter_; }
    virtual value_type& get() const {
       return *iter_; }
  protected:
    virtual const std::type_info& type() const {
       return typeid(iter_); }
    virtual bool up_compare(
       iterator_base* rhs) const
    {
      holder* r = dynamic_cast<holder*>(rhs);
      return iter_ == r->iter_;
    }
  private:
    iterator_type iter_;
  }; // class holder
}; // class iterator_holder
```

That's quite a chunk of code, I don't intend to dwell on the details. The main thing to note is that the outermost class **iterator_holder** is parameterised on the value type, on what the held iterator points to, rather than the type of the iterator itself. The holder and **iterator_base** are the template class with non-template base described above.

We can rewrite filter_iterator to use **iterator_holder**, and our example becomes:

```
filter_iterator<int, Even> fb(vec.begin(),
   vec.end(), Even());
filter_iterator<int, Even> fe(vec.end(),
   vec.end(), Even());
for( ; fb != fe; ++fb)
{
  std::cout << *fb << std::endl;
  ... do something else with *fb ...
} // for ...
```

But look at this. We can also write this:

```
std::deque<int> deq;
...
filter_iterator<int, Even> fb(deq.begin(),
   deq.end(), Even());
filter_iterator<int, Even> fe(deq.end(),
   deq.end(), Even());
for( ; fb != fe; ++fb)
{
  std::cout << *fb << std::endl;
  ... do something else with *fb ...
} // for ...
```

Exchanging the **std::vector** for a **std::deque** doesn't require any other change. That's rather a useful result.

One down, one to go

While using the **filter_iterator** now requires less typing, it still doesn't have the runtime behaviour we want. While the dependency on the wrapped iterator's type has been removed, the predicate's type still intrudes. Again, however, we are not interested in the details of the predicate type. It isn't exposed through the public interface, and internally we only care that it has some function that takes a **value_type** and returns **bool** – we really don't care about the precise details. By writing a similar **predicate_holder**, the **filter_iterator** reduces to that shown in Listing 6.

It doesn't look hugely different from the first version, but the type signature is much more straightforward. Not only is it much easier to work with, it's significantly more flexible. (Listing 7.)

Victory is mine, I gloated to myself before realising I'd rather exceeded my brief. The **iterator_holder** class, far from being an implementation detail for the **filter_iterator**, is a useful piece of kit in its own right, providing a runtime polymorphic type-safe wrapper for arbitrary iterators. It's the thing, the little bit of scaffolding, I should have been aiming for to begin with. With that in place, providing the runtime behaviour I need, everything else – filtering iterators, transformers, or whatever – can be built on and with it.

Further reading

'External Polymorphism – An Object Structural Pattern for Transparently Extending C++ Concrete Data Types'. What we've just done has a name. http://www.cs.wustl.edu/~cleeland/papers/External-Polymorphism/External-Polymorphism.html

Googling around I found 'A Fistful Of Idioms – Giving STL Iterators a Base Class', an article from *Overload 38* back in July 2000, in which my chum Steve Love develops an **any_iterator** wrapper class. He approaches it from a slightly different direction and rather more formally, but our results are similar. I must have read it at the time, but can't explicitly recall doing so. If you don't have that to hand, it's available in the members area of the ACCU website at http://accu.org/index.php/journals/479.

Towards the end of last year, Thomas Becker published on article on Artima, http://www.artima.com/cppsource/type_erasure.html, entitled 'On the Tension Between Object-Oriented and Generic Programming in C++ and What Type Erasure Can Do About It' in which he developed **any_iterator**, a type-safe, heterogeneous C++ iterator. The Adobe Source Libraries also have an **any_iterator**, (see http://opensource.adobe.com/classadobe_1_1any__iterator.html). All of those are rather better developed that what I stumbled over writing **filter_iterator**. Perhaps I should pre-emptively google next time?

Boost.Iterator includes a **filter_iterator**. It also includes several other useful iterator adaptors, including a **transform_iterator** and a **zip_iterator**. Matthew Wilson describes a bidirectional **filter_iterator** in an extract from his *Extended STL* book available at http://www.informit.com/articles/article.aspx?p=770642&seqNum=5. I found another at http://www.salilab.org/~drussel/pdb/iterator_8h-source.html. All are essentially equivalent to the naive version presented above, if more complete and, in the Boost case, more formally specified.

On a slightly different track, a chap called Mr Edd describes an **opaque_iterator**, designed to reduce compile-time dependencies, at http://www.mr-edd.co.uk/?page_id=43.

Listing 6

```
template<typename value_type>
class filter_iterator
{

public:
  template<typename iterator_type,
     typename predicate_type>
  filter_iterator(const iterator_type& begin,
                  const iterator_type& end,
                 const predicate_type& pred):
    current_(begin), end_(end), pred_(pred)
  {
    while((current_ != end_)
       && (!pred_.test(current_.get())))
      current_.advance();
  } // filter_iterator
  filter_iterator(const filter_iterator& rhs) :
     current_(rhs.current_), end_(rhs.end_),
     pred_(rhs.pred_) { }
  filter_iterator& operator=(
     const filter_iterator& rhs)
  {
    current_ = rhs.current_;
    end_ = rhs.end_;
    pred_ = rhs.pred_;
    return *this;
  } // operator=
  value_type& operator*() const {
     return current_.get(); }
  value_type& operator->() const {
     return current_.get(); }
  filter_iterator& operator++() { advance();
     return *this; }
  filter_iterator operator++(int)
  {
    filter_iterator c(*this);
    advance();
    return c;
  } // operator++
  bool operator==(const filter_iterator& rhs)
     const { return current_ == rhs.current_; }
  bool operator!=(const filter_iterator& rhs)
     const { return !(operator==(rhs)); }

private:
  void advance()
  {
    do
    {
      current_.advance();
    }
    while((current_ != end_)
       && (!pred_.test(current_.get())));
  } // advance
  iterator_holder<value_type> current_;
  iterator_holder<value_type> end_;
  predicate_holder<value_type> pred_;
}; // class filter_iterator
```

Listing 7

```
filter_iterator<int> fb(vec.begin(),
   vec.end(), Even());
filter_iterator<int> fe(vec.end(), vec.end(),
   Even());

for( ; fb != fe; ++fb)
{
  ... do something with *fb ...
} // for ...

// and now the odds
fb = filter_iterator<int>(vec.begin(),
   vec.end(), Odd());

fe = filter_iterator<int>(vec.end(),
   vec.end(), Odd());

for( ; fb != fe; ++fb)
{
  ... do something with *fb ...
} // for ...
```

Kevlin Henney's article in the August 2000 *C++ Report*, 'Valued Conversion', covers the same techniques describing a class which can hold any value. That code grew up to become **`Boost.Any`**.

Thomas Guest has fun building iterator pipelines in 'Zippy triples served with Python', at http://wordaligned.org/articles/zippy-triples-served-with-python ■

Notes

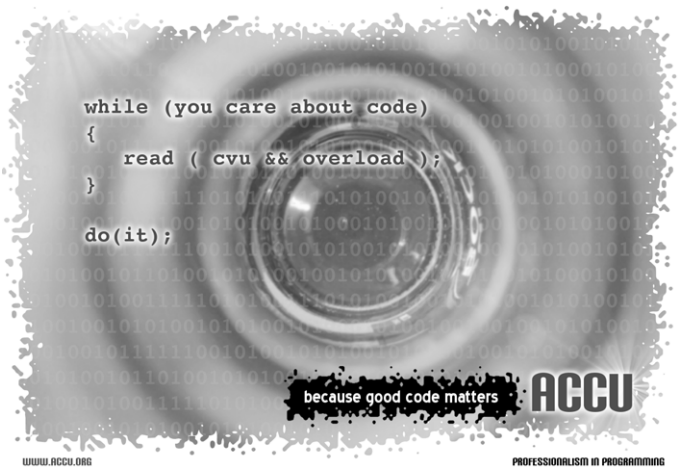
[1] Finding the Utility in a **`java.util.Iterator`**. The slides are at http://www.jezuk.co.uk/accu2007/iterator/

[2] Slides from the presentation are available at http://www.jezuk.co.uk/files/iteration/, so you can compare and contrast.

[3] These are the requirements for an input iterator. The C++ Standard Library describes five distinct iterator concepts.

Acknowledgements

I would like to thank Thomas Guest for his many contributions to this article, from graciously allowing me to steal his examples for my presentation through to his many suggestions and improvements to the article itself.



ACCU Conference 2008

Jez Higgins and friends look back at this year's ACCU Conference.

Every year my recovery from the conference takes longer. I'm not talking about the physical effects of perhaps too much late night talking and drink and perhaps too little sleep, I mean the time it takes from my brain to stop churning seems to be extending and extending. Maybe I'm just getting old, but I prefer to put it down to the excellence of the programme. It was as wide and as varied as ever, mixing programming techniques, design, management, methodology and more, along with a generous portion of functional programming. This year's keynote speakers were Tom Gilb, an authority on software project management, Simon Peyton-Jones, inventor of Haskell, Andrei Alexandrescu, noted C++ expert, and Roger Orr, a member of the BSI and ISO C++ panels and well known ACCU member. Sessions covered topics from debugging to the building of Cambridge colleges, from Python on .NET to project management, from parallel processing to Lisp, from robots to a search for God in the machine. The extra-curricular entertainment included Just a Programming Minute, 'birds of feather' sessions, technology demonstrations, the odd game of squash, a competition to win an X-Box, and a very great deal of socialising (by which I mean talking and drinking late into the night). It's really no surprise that many delegates go home with over-heated brains and slightly queasy tummies.

Last year, I thought the programme was ridiculously good. This year's was better (or, depending on your outlook, worse). There were several occasions when I was interested in seeing four out the five sessions, and there wasn't a time when I was interested in less than two. It was ludicrously hard to decide, and I know from asking around that I was far from alone in finding it difficult. Giovanni Asproni, our conference chair, deserves hearty congratulations, right before we take him round the back and give him a good roughing up. Next year I'm shooting for a 15-day single track conference, just to make the choices easier on everyone.

To give a flavour of things to those who couldn't attend, to jog the memories of those who were there and are still recovering, and to encourage those who are thinking about next year, here are some write-ups from various attendees. These are little tastes of what people felt about the event, summaries of some of what went on, and synopses of sessions attended.

Allan Kelly <allan@allankelly.net>

It's always troubled me that we don't have more Java and C# at the conference but I think I understand now. The kind of people who come to ACCU like a certain type of programming challenge, you find this in C++ but not so much in Java or C#. This is partly because Java and C# do their job well and partly because they are, well, shall we say, a little boring. They do what they are supposed to. Because of the way C++ has developed and the type of applications it is used for, it includes more of these challenges.

(Having said that, I had a conversation the other day which started: Java is not as interesting as C++, in fact Java is boring, but reflection is interesting. And annotations. And the libraries, and ...)

It seems functional languages too have these challenges. People here are very excited about all things functional. Already it looks like next year will have plenty of Haskell, Erland, Lisp etc.

Oddly, for the first time ever I found myself with spare time on my hands at the conference. Not a lot but there was the odd session where I was not interested in anything – although there were more sessions where I wanted to be in two places at once. I think this is more a reflection on myself, I'm increasingly post-technical and the technical sessions here were hard core.

Some more comments, observations and thoughts, with no particular linking theme ...

Jobs and banks

There were a lot of bankers at the conference this year – or rather developers who work in banks. In truth this is always the case but this year I think there were more. It also gave the opportunity to find out what was happening in the financial job market.

Before Christmas I think many people were expecting a shake-out similar to that of 2001/2002 but it seems the effect of the crisis or credit crunch is very mixed. I observed a few weeks ago that one effect has been to push up rates and while some banks do seem to be shedding staff others are still hiring and demand is strong. I talked to someone from one (American) bank which had drastically cut back staff, while people at another (English) bank were still hiring lots, and another (Scottish) bank which recently bought another (Dutch) bank is having to hire lots of people to help integrate the systems of the two banks.

Lies, damned lies and statistics

I've had a downer on metrics and statistics for a long time. But I've also been aware that getting the right data and measuring the right numbers is often the key to success. Tom Gilb is firmly in the numbers and metrics camp and argues a good case for measurement and targeting. I was lucky enough to spend lots of time listening to Tom and talking to him.

From the Conference Chair

Giovanni Asproni <aspro@acm.org>

This year was my first one as conference chair, and I'm happy to say it was another great event. I wish I could take all the credit, but, unfortunately, I can't ;-) . In fact most of it goes to the conference committee – Ewan Milne, Alan Lenton, Francis Glassborow, Tim Penhey, and Aaron Craigie – to Kevlin Henney (in the role of special adviser for the committee), and to our conference organisers Julie Archer and her colleague Marsha Goodwin, who, as always, did an outstanding job.

Some highlights include the opening keynote from Tom Gilb, which caused a bit of a stir and a heated discussion between Tom on one side, and Nico Josuttis, Jutta Eckstein and Peter Sommerlad on the other; the special track on functional programming, which was a sell out – every single session (including the Erlang pre-conference and Simon Peyton-Jones' keynote) in the track had a huge attendance; and Jon Lakos' session, in which he managed to present more than 560 slides in 90 minutes establishing a new world record that won't be easily beaten

Unfortunately, we had also some negative feedback – many delegates kept complaining that, in every slot, there were too many good sessions at the same time, and they didn't know which one to choose. The only answer I can give them is that, in this respect, we are already working hard to make the 2009 edition much worse!

Finally, there have been a few changes in the committee. Aaron Craigie, unfortunately, had to leave due to other commitments, but we also have three new members; please join me in welcoming Astrid Byro, Roger Orr, and James Slaughter.

It seems that, as with so many other things in life, the key is doing it right. It is very easy to set the wrong targets, to measure the wrong thing and produce side-effects you don't want. But when you look at the right numbers, measure the right thing, and set the right targets you can get great results.

But it isn't easy, most people get it wrong.

Tom is a fascinating guy, if you ever get the chance to hear him speak do so. A lot of his ideas come directly from Deming – who he knew personally. I don't think it is too much of an exaggeration to say Tom may be the Software World's own Deming. If nothing else Tom interprets Deming's message for software development.

that means 85% of us are working on failing projects

Software development success

In an aside Tom also pointed out that the UK now has Royal Academy of Engineering and that they (not so) recently produced a report on software development. It's a shame that you can't download the report, I'd like to read it.

I notice the comment that 'only around 16% of IT projects can be considered truly successful'. That 16% seems very close to the figures given in the MIT Sloan Review piece on the IT alignment trap last year. That report said 7% of companies had effective IT departments which delivered on business objectives and another 8% who were effective but were not aligned with the business.

Unfortunately that means 85% of us are working on failing projects. Depressing.

The 1968 wrong turn reconsidered

I've been heard to say that somewhere about 1968 the software industry took a wrong turn. We went down the route of engineering, planning and tools rather than people and learning. But the conference made me wonder, maybe it wasn't such a wrong turn, maybe it was a diversion we needed to take so we could solve some problems. Now those problems are kind of solved we need to refocus on the people.

Maybe.

Product managers

I have long claimed that UK business do not get Product Managers, this might now be changing. The term Product Manager was in wide use and more people seemed to have a Product Manager on their team.

Lets hope I'm right.

Still, there are too few Product Managers, too few of them are really good, their role is still misunderstood and there is not enough training for them.

Next year's conference

I'm no longer on the committee for the conference but I still have conversations about it. It is already taking shape in people's heads and promises to be an even better conference. For better or worse the conference is unlikely to get any bigger. If it were to get bigger it might lose some of its flavour.

I think the next two or three years will be an interesting time in the UK and European conference scene. The arrival of the profit making QCon is having an effect and I know there is some debate about the future, style, content and so on of other conferences. At a guess I think you might see one or two new conference appear and others (perhaps) disappear or change.

Tim Pushman <tpushman@gnomedia.com>

So it's April, I'm in Oxford... it must be the ACCU conference again. I got off to an inauspicious start by spraining my ankle before arriving and I've spent the conference hobbling around. Unfortunately, many of the talks were upstairs, slowing me down even more.

Day one

The first day's talks cover a variety of subjects: network services and programming, agile development, programming methodology and robotics.

For the morning session I went to Roger Orr's talk on 'Programming in a Networked World', where he showed us some of the pitfalls of writing programs that had to function over network connections. The talk was quite high level and more an overview of the possible problems of latency and bandwidth issues, but unfortunately didn't have time to cover solutions except in a general sense.

I chose the afternoon track on robots, something that sounded fascinating and unusual. Bernhard Merkle started with an introduction to his work at Sick AG, a German company that makes sensors, essential to any robot. Indeed on the control side, a robot is a collection of sensors (input devices) and motors (output devices) and the coordination between them. After a quick look at what his company has been doing as part of the DARPA Urban Challenge, he then took us into the depths of Microsoft's Robotics Studio, a complete development environment and simulator for creating robotic devices. It's big and it's complex, and it's probably very useful for a large enterprise development, but it seems overkill for someone just starting up with robots.

Which led nicely into the second robotics session, from Ed Sykes and Jan-Klaas Kollhof. They had brought in a couple of Lego Mindstorms robots and we watched them motor around the floor while Ed and Jan explained how they were programmed to do what they did. They had each taken a different approach to the programming, Ed using Microsoft Robotics Studio, and Jan using an open source environment for similar ends. While MSRS allowed Ed to create and control his robots through a graphical interface and then simulation testing, Jan took the approach of writing external scripts to control the robot. Firstly through using Python scripts running on his laptop and communicating via bluetooth and next using NXC and pblua to compile and push the code on to the robot, making it run autonomously. Lego robots provide a very open and flexible environment for playing around with this sort of thing and it was impressive how little coding it took to have them following a line on the floor. With addition of a camera to one of them, it could then follow the other robot, which had an orange ball on top as a identifier, around.

Day two

One of the tracks at this conference has been on functional programming and today's sessions covered FP in general and more specifically, Erlang and Haskell. Every conference has a special track, in the past there has been template programming in C++, C# and .Net programming, or open source software. The special tracks often are on subjects which are not mainstream at present. But if they are well attended, it's usually an indication that they may be in the next couple of years. And the FP sessions were very well attended (I spent a couple sitting on the floor) so expect to see more of these languages in the future.

The Erlang session, by Joe Armstrong, covered the origins of Erlang in the telecoms industry. The industry needed a system that was very robust (99.9999999% was claimed), could be updated while running and was fully concurrent. Erlang was the result. Interestingly, it creates and manages its own processes, allowing process creation to be very fast and cheap. Each process is a service and the system uses message passing between the services to provide robustness. This also provides scalability over multiple servers/CPUs and very robust error handling and recovery. Joe is the inventor of Erlang and knows his subject inside out. One of the more interesting bits of information was how Erlang allows an application to easily scale over multicore processors and as multicore becomes more common, then Erlang could become the language of choice for developing applications in the future. Most applications today only know how to work with single core CPUs and once we move to having hundreds of cores in a processor, much of the CPU could be wasted.

The Haskell session from Simon Peyton-Jones was so full that every available spot was occupied. Simon gave an excellent talk, spread over two sessions, on the basics of Haskell, and walked us through some of the implementation details of XMonad, an X server window manager, written in about 500 lines of Haskell code. Personally, it's not easy to get my head around the programming style of Haskell (it reminds me a bit of Prolog for some reason) but the ability of Haskell to enforce programming without side effects (another session was devoted to this subject) was in itself very interesting. As far as I can see, it still an academic language, but many of the ideas behind it are sure to find themselves into mainstream programming.

Day three

Interesting session this morning, from Schalke Cronje, on RPM package management. One of those sessions that cover a subject I know something about but have never had time to look at in depth. Schalke covered the basic command line usage of RPM and then showed us how to create spec files and build packages that could be distributed over multiple platforms. Nothing stunningly new, but all explained clearly and it will certainly give me the confidence to try packaging up some of my stuff with a few more deployment features.

Later we had a BoF (Birds of a Feather) meeting about the ACCU website, bringing together any members who had input on the ACCU website and what direction it should be heading in. Lots of good ideas, and there probably isn't time to implement all of them, but we can make a list, put the simple stuff into practice and then work out how to manage the more advanced ideas. About a dozen people attended up, which was a good showing. Stay tuned for more information and updates on this.

Later I went to a session by Astrid Byro on using Documentum to manage the documentation for a large EU organisation. A great overview of how an enormous paper consuming and producing machine like the EU handles its documentation. Or rather, how it would like to handle it if they ever put some of this into practice. There seem to be few organisational changes needed as well as purely technical matters.

Day four

First session today was called 'Is FP for me' and covered the areas in which functional programming might be useful. The session, given by Hubert Matthews, provided an excellent overview of when and where to use a functional programming language. He started by discussing the different types of language, using as examples Fortran (for mathematical calculation), COBOL (for business applications) and Lisp (for more algorithmic and abstract programming). Although all three languages were created in the 1950s, they are still with us today, COBOL having inspired the creation of imperative languages such as C/C++ and Smalltalk, Lisp providing ideas which have developed into Haskell, OCaml and similar. When asked which language would give a neophyte a good introduction into the techniques, Hubert suggested Haskell.

Second session, from Didier Verna, was on Lisp programming and gave an interesting look at the state of the art for Lisp in today's world. Didier is passionate about Lisp and energetically set us straight on some of the wrong impressions we might have had about Lisp, such as performance, strong/weak typing, the object system (CLOS) and optimisation. I first played around with Lisp many years ago and there have been a lot of improvements since then. I'm tempted...

It was a very interesting conference and I took the opportunity to try and attend all the sessions on functional programming and see if I could figure out what it meant and what it was useful for. I can't think of any projects that I could use it on at the moment, but some of the ideas I've heard are very useful in any programming toolkit, such as 'no side effects' in functions. As I mentioned in an earlier, the specialised tracks at ACCU are often on subjects that we can consider 'emerging technologies', in that they are not mainstream but where there is a growing interest in them. When there is strong interest in the tracks, as there were in this years ACCU, then it's a pretty good indication that this is a technology that is set to become more mainstream in the next years. Keep an eye on Functional Programming.

the specialised tracks at ACCU are often on subjects that we can consider 'emerging technologies', in that they are not mainstream but where there is a growing interest in them

Anna-Jayne Metcalfe <anna@riverblade.co.uk>

A functional workout

If you've not come across it before, Erlang is a functional language designed for concurrent programming. For someone from an object orientated background it is quite a paradigm shift, and the syntax takes some getting used to. Nevertheless, it is pretty obvious to me already that this is a language with some real strengths.

One thing I didn't realise during our preparation for the conference was that Erlang was developed from Prolog – which may explain why parts of it (pattern matching, for example) seemed strangely familiar. I studied Prolog as part of a 'Machine Intelligence' course at Surrey University.

Haskell and Microsoft's latest research language F# are aimed at the same problem domain. It will be interesting to see how strong the take-up of such functional languages is over the next couple of years, and whether we see the start of a longer term trend of increasing adoption.

Having said all that, as a (primarily) user interface developer I have no idea what practical use it is likely to be to us in the immediate future...but of course you never know...

Value Delivery for Agile Environments [Tom Gilb]

The thrust of this session was that although agile methods are better at organising development tasks than conventional methods, they do not really focus on the needs of stakeholders. For example, they do not provide guidance on the business value of each potential task. By contrast, Evolutionary Project Management (EVO) is more focused on business goals than tasks and iterations/sprints. An approach such as EVO can be used together with agile approaches such as Scrum to great effect.

EVO is based on continuous measurement and reassessment of business metrics, stakeholder requirements, budgets, goals, impact estimation (e.g. via impact estimation tables), estimating, planning and tracking. Key principles include:

- Critical Stakeholders determine the values a project needs to deliver
- Values can and must be quantified numerically (no matter what it is, the chances are somebody has measured it in some way. It is critical that agreement is reached on how individual values are measured).
- Values are supported by a Value Architecture (defined as 'anything you implement with a view to satisfying stakeholder values').
- Value levels (the degree of satisfaction of value needs) are determined by timing, architecture effect and resources.
- The required value levels can differ for different scopes (e.g. where, which stakeholder). Setting value levels too high can kill projects by delaying delivery and inflating costs.
- Value can be delivered early. Plan to deliver real value to stakeholders as early as possible, and continue to deliver additional value continuously.
- Value can be locked in incrementally – deliver production quality systems throughout, and not 'quick fixes'.

- New values can be discovered by stakeholders in response to delivered values. It therefore follows that developers must be in direct contact with stakeholders.

My initial reaction was that EVO in its pure form may not be entirely suitable for a small ISV due to the sheer quantity of analysis required; however this is no different from the situation with any process/methodology – Scrum (for example) doesn't work particularly well in a micro-ISV environment either. The lesson is to take the good bits, and leave those which bring in more overhead than you need. That said, Tom apparently has a case study involving a 3-person team which isn't too far removed from the micro-ISV world.

Either way, EVO is definitely an approach professional developers and project managers should be aware of. The majority will of course carry on in blissful Waterfall-esque ignorance as always...

Santa Claus and Other Methodologies [Gail Ollis]

Gail is an active member of the ACCU South-Coast group, and a very entertaining and thought provoking speaker.

'I don't believe in methodologies'

Methodology is strictly the study of methods etc. rather than their application, but the use of the name in conjunction with development processes can (unfortunately) lend them 'instant' credibility in the eyes of some – the 'follow this and everything will be perfect' delusion. The real world is not like that – any 'process' is only going to work well if you buy into it and tailor it to your own needs. If you follow a process blindly, it will almost certainly fail you.

Gail followed her introduction with a brief historical foray into a long dead software development 'methodology' called RTSAD, and a project development process called Goal Directed Project Development (GDPM), outlining the failures of both when applied within an organisation to illustrate her point.

New methodologies offer new buzzwords, which can lead companies to adopt them for the wrong reasons. Particular groups of people seem to be most susceptible to this:

- Budget holders
- Seekers of the 'One True Way'
- Advocates of the 'latest big thing'
- Grand planners

(the first and last are often managers; the second and third are often developers).

At the end of the day, although these are people problems – and not process problems – persuading people to change the way they work is all too often exceptionally hard.

The lesson is not to look at the solution (e.g. 'adopting <Methodology X> will solve all our problems'), but at the real problem. Once the problem has been identified, potential solutions can be visualised and investigated. Some questions we could (for example) ask about a potential solution include:

- How does this address our specific problem?
- What does this step/artifact/process do for us?
- What demands does it make of us?
- Can we integrate this step/artifact/process and its tools smoothly with what we have?
- Does it impede continuous improvement?

As ever, there is (unfortunately) no magic bullet.

Robots Everywhere [Bernhard Merkle]

We met Bernhard for the first time last year when he ran a very interesting session on architectural analysis tools. This year he has turned his hand to looking at the world of robotics.

Bernhard started the session with a fascinating illustrated summary of the state of the art today, including competitive events such as RoboCup (robot football) and the DARPA Grand Challenge (autonomous vehicles).

Concurrency and (naturally) functional programming are fundamental to robotics. Although there are a number of established players in this field, Microsoft are now targeting the emerging home and educational markets with Microsoft Robotics Studio (MSRS) and the parallel computing initiative.

Microsoft apparently learned that typically 80% of the development time on robotics project is currently being spent on developing limited use frameworks, and the MSRS effort is in part aimed at generalising these sorts of efforts. A secondary aim is obviously to support adoption of the .NET Framework within robotics applications. MSRS has a heavily concurrent and distributed architecture, which Bernhard spent some time describing in depth. It was also interesting to see that C# was being used rather than the (I would have thought) more well suited F# functional language.

All in all this is a fascinating subject, and no doubt one which will become more and more prominent.

A Tale of 2 Systems [Pete Goodliffe]

This session looked humorously at the long term impact of design on a software system, using two real examples. Pete's assertion is that the quality of a project is determined mostly by the quality of its design.

Good designs should be:

- Easy to modify
- Easy to extend
- Flexible enough to accommodate change without stress
- Fit for purpose
- Easy to understand

Pete explains a tricky problem through the medium of interpretive dance

Pete gave examples of two similar systems he had worked on to illustrate these principles:

The Messy Metropolis This was a spaghettified mess, the code for which had grown 'organically' over time with very little thought. Pete rather appropriately illustrated it with a picture of a turd! We've all seen systems like this, so I'm sure I don't need to elaborate further ...

Eventually, such systems grind to a halt and effectively force a complete rewrite – whereupon the cycle can all too often repeat, and at huge cost. Design problems can be caused by company culture (e.g. empire building, not giving developers time to rectify smells in the design) and poor development processes with insufficient thought given to design issues. Pete ably described the problems this particular projects caused within the company at every level from support to sales, marketing, customer support and manufacturing. It (not surprisingly) eventually ended up in a costly rewrite – which is a high risk proposition in its own right.

Design Town This project was different from the outset. The project was run by a small, flat team with a clear roadmap and following a defined process (XP in this case, but we won't hold that against them.).

Perhaps crucially, the design was limited to that which was sufficient to meet the requirements (a key agile principle, in my view) without attempting to include detailed provision for possible future requirements. In this system, the design made it far easier to add new functionality. It was straightforward to locate where specific functionality lay, and new functionality gravitated naturally to the right place. Bugs were also easier to locate and fix. Most importantly, the software developers took responsibility for the design. This last point is (in my view) fundamentally important – some developers I meet are sadly lacking in the essential motivation to do this.

So, what lessons can be learnt from these two projects?

- Design matters, but it does not happen without conscious effort
- People are key (this touches on Gail's session earlier)
- The team must be given (and accept) responsibility for design
- Good project management

How then can we improve a bad design?

- First of all, we can't improve it unless we understand it. There is always information in SCC, documents etc. which can reveal aspects of the history of a project, so why not go digging and see what you can find?
- Describe the process which seems appropriate to deal with the state of the existing design (run away, re-write, refactor etc.)
- Plan a new design based on the requirements and constraints we know now (as opposed to those we thought we knew at the outset)
- Plan a roadmap for how to take the codebase to where we want to be, and continuously refine it as you proceed along the route.

May You Live in Interesting Times [Andrei Alexandrescu]

This session was a humourous illustration of the ideas and issues involved in the C++ 0x language design, and how tricky it can be to design a modern language.

Andrei illustrated that in such a large language there are so many domains, that no one person is likely to be an expert in all – and C++ is such a big language that this is almost inevitable. Even the most simple problem – writing an `identity()` function which returns its value – is not as simple as it seems in C++ if all use cases are considered.

He also described some of the more notable new language features in C++ 0x:

- Higher order functions
- Closures
- Lambda functions (a recent addition to C++ 0x). If you use functional languages you will appreciate the significance of this!
- Variadic templates (templates with variable parameter lists)
- Types of types (which introduce structure to types and allow type interfaces to be documented)
- Concepts
- Threads (based on `boost::threads`)

The bottom line is that if you work with C++ code and haven't taken a look at what is coming in C++ 0x, you probably should...

C++ Refactoring [Peter Sommerlad]

This session focused on TDD and C++ refactoring in Eclipse. Peter's group at the Institute for Software has produced some very interesting C++ refactoring and unit testing plug-ins for Eclipse CDT. We have been talking to Peter about static analysis tools for Eclipse during the week, so this was a great chance to see the tools his group have developed in action.

Peter gave a brief introduction to TDD for anyone who wasn't too familiar with it, before firing up Eclipse to demo the CUTE plug-in. At first glance, the plug-in seems similar in concept to TestDriven.NET, but with a better user interface. For example, it has a comprehensive tool window (a little reminiscent of the NUnit GUI) which shows not only the tests but the console output from the tests themselves. One very nice feature of the CUTE plug-in is that it will generate stub tests and test suites within the IDE automatically.

Peter spent most of the session going through a couple of examples using the CUTE plug-in. Unfortunately we didn't have time to look at the refactoring plug-in in depth, but what we did see certainly looked quite comprehensive – possibly more so than that provided for Visual Studio by Visual Assist.

Seven Deadly Sins of Debugging [Roger Orr]

Roger is a member of the ISO C++ Standards Committee, and a specialist in the field of debugging. Having attended one of his sessions last year, we had a pretty good idea that this keynote would be both entertaining and informative.

Roger started by stating the obvious – that the best bugs are those which do not occur, and that by learning to apply techniques to reduce problems up front (e.g. good design, unit testing, code analysis, defensive programming etc.) we can reduce the risk of bugs occurring. None of this should be news to anyone attending the conference. After using such techniques to remove the obvious bugs, we are left with everything else. Debugging is quite obviously here to stay.

It has been stated that better programmers can be 20 (?) times better at finding bugs, spend less time fixing them and put fewer new bugs in by doing so. The obvious question this then raises is 'Why is there such a differential, and what prevents so many of us from learning?' Enter the Seven Deadly Sins of Debugging.

Inattention can lead us to not look closely enough at what we are doing, miss the obvious patterns ('what are the real symptoms of the bug?'), and repeat the same mistakes again and again. Details are very important in debugging – logfiles, configuration information etc. can all yield crucial information, so the more information which can be automatically generated the better. Collecting this information up-front can also save you from having to generate the information you need while actually investigating the bug.

Debugging also requires very focused concentration, so taking adequate breaks is essential. There is nothing less productive than staring at a debugger with a deadlocked or clueless mind– and yet all too often developers attempt to debug in exactly that way.

Keeping checklists (e.g. our own lint configuration triage procedure) can also help greatly, since it is all to easy to miss something obvious when you are under pressure to fix a critical bug. Similarly, the insight afforded by a second pair of eyes can also help, so we should never be afraid to ask for help.

The corresponding virtue is observation, which leads us to ask interesting questions such as:

- What is our strategy for observing program behaviour?
- What tools are available to give us the information we need?
- How can we make this easier at the design stage?

Pride can lead to higher quality code in the first place, but when mis-applied it can also unfortunately:

- Prevent us from asking for help when we are trying to fix a bug
- Lead to a refusal to admit that a bug is our problem (and because it isn't our problem we won't look for better ways to prevent bugs).
- Keep us following a wrong debugging hunch rather than stepping back and re-examining the evidence.
- Lead to inappropriately clever code, and lead us to writing things from scratch when we should reuse existing solutions.

The opposite of pride is humility. Questions such as "I could be wrong", "What have I missed?" and "Who can I ask, and how?" can lead to the insights you need to fix that troublesome bug.

Naivety tends to prevent us from learning from our mistakes, and lead us to make mistaken assumptions about where the problem lies. On the plus side, the simplest fix for a bug is likely to be the right one. The corresponding virtue is wisdom, e.g. standing back to reflect on how the bug happened, why, and how we can prevent it happening again.

Anger needs no introduction. It can cloud our judgement, cause us to miss obvious clues, and to deny the implications of the evidence we have.

Sloth can lead us to try to avoid 'unnecessary' work while we are writing code in the first place. When the resultant bug surfaces, we poke around in the debugger in vain. It also results in ignorance – a failure to read around

the subject or fully understand the technology. Sadly, this is all too common.

The corresponding virtue is diligence – by learning enough about the system to understand how it behaves, we dramatically increase our chances of identifying the cause of bugs in a timely manner.

Diligence also leads to other positive effects – for example spending time upfront to save even more time later. By writing scripts, adding logging etc. we can often make a real difference when investigating a bug. Another often overlooked technique is to make error codes unique enough to look up in a search engine.

Blame – As the saying goes, a bad workman blames his tools, users, tests, third-party components, ... anyone but themselves! Blame doesn't fix the problem, but may lose you some allies. Even if you can blame another system, you still have a bug to fix. The corresponding virtue is quite obviously responsibility.

Vagueness is fatal to effective fault finding. 'What exactly is the bug?' and fixing a bug, but not 'the' bug can both intervene to mess things up. However, precision greatly improves bug hunting. If something seems to be breaking repeatedly, focusing on what you are doing, making error messages more useful, and so on can all help.

The bad news is that debugging is hard, and is not likely to get any easier:

- There are more distributed systems
- The trend is towards an increasing mix of technologies and languages
- Higher security requirements
- More dependencies and faster time to market

The more effective we can be at preventing, identifying and fixing bugs the less time we will spend unnecessarily in front of the debugger.

Researching a Problem and Getting Meaningful Results [Alan Lenton]

If you're on an obviously failing project, how do you get management to listen?

That was the question posed by this session. One obvious answer is to quantify it in a form they understand and will therefore listen to. This actually dovetails rather closely with Tom Gilb's EVO session earlier this week, albeit from a different perspective.

Fortunately, with a bit of work you can quantify just about anything (technical debt anyone?). There is however a danger that by quantifying things doing so becomes an end in itself, rather than a tool to solve a problem. Once you quantify a problem, the presentation method of choice for managers is the spreadsheet, which also provide a simple way to present the results graphically if appropriate.

A financial cost estimate is key for this target audience. Once you have an idea of how long an issue would reasonably take to fix, it is straightforward to calculate this based on time to fix and hourly cost including (or excluding, for maximum impact when you add them in later!) overheads.

If you are planning to make a financial case it is also worth remembering that capital costs and labour costs do not always compare directly, since the former can (certainly in the UK) have an impact of profit margins but the latter will not (you find this sort of stuff out when you set up your own company, believe me!).

A key question is how to quantify a failing project, rather than just one part which can be fixed? The obvious metric is 'how much is the company spending per month on this project?'.

The Complete Guide To C++0x [Alisdair Meredith]

Alistair Meredith of Codegear is a member of the C++ Standards Committee, and this session was a lightning tour of the changes in C++ 2009 (otherwise known as C++ 0x. Alistair stated that they are aiming for a 2009 release – the first full C++ standard release since C++ 1998. As such, it is a major update.

Alistair first of all described the features which will (unfortunately) be missing from this release: e.g. library features beyond TR1, C++ modules, maths binding, and garbage collection have been deferred until TR2 (due in 2012?) or will be incorporated into separate standards.

The final release candidate of the standard should be out in September 2008 – which would mean that all comments will be received by January.

So what's new? In short:

- 50 new language features
- New libraries
- A wider set of standards
- Revisions to existing libraries
- The incorporation of features from C99 + TC1 + TC2 + Unicode TR
- ECMAscript regular expressons
- Threading

Some of the most fundamental changes are (as is to be expected) in the area of concurrency. Notably, C++ 2009 will finally define a modern memory model, which should lead to less uncertainty in defining what is and is not acceptable in multi-threaded code. The biggest impact of this change is in defining which fundamental assumptions can and cannot be made by optimisers, so it should be largely transparent for most.

Other changes in this area include the addition of defined atomic operations (there is a new atomic keyword), intrinsic threads and locks, and (possibly) futures. Thread pool support has been deferred to TR2, which is a shame but understandable given the volume of change already proposed.

Alistair talked at length and in detail about the new and changed language features, but did not have time to discuss the corresponding library changes. I can't even begin to do everything justice, so here's an editted list of the changes he described:

- The meaning of the `auto` keyword has been changed to a type deduction specifier (a.k.a. dynamic languages).
- Template aliases (non specialised template typedefs)
- Raw string literals
- UTF8 string literals
- Delegating constructors (allows constructors to delegate object initialisation to another constructor)
- Inheriting constructors
- Lambda expressions
- `nullptr`
- Variadic templates
- Perfect forwarding in templates (deals with the explosion of overloads where const is involved)
- Move semantics, through `rvalue` references
- ... and 40 more ...

And that's just the compiler...!

The State of the Practice [Tom Gilb, Hubert Matthews, Russell Winder, Peter Sommerlad and James Coplien]

The subject of this panel was in effect: 'Are we barking up the right tree? So many developers have no idea of basic good practice. Discuss..'

While I can't even begin to do the ensuing discussion justice, the responses of the panel members to the opening question give an interesting insight into the discussion :

Tom Gilb: 'There is not enough focus on delivering value to our stakeholders.'

Hubert Matthews: 'We have forgotten the human element and reward structures reflect that.'

Russell Winder: 'Polarisation. There is (unfortunately) a lot of dross out there.'

Peter Sommerlad: 'The state of practice is partly a reflection of past failure in academia. It is now too easy for lay people to produce badly written software.'

James Copelien: 'This is a wicked problem without clear cause and effect.'

Garry Bodsworth <garry.bodsworth@gmail.com>

Disclaimer: These are my interpretations of what I learnt from the talks rather than a transcription of what they said. This means that I probably misheard and misinterpreted some parts which may be hazardous to your health.

My employer, DisplayLink, were very generous and allowed me to attend the ACCU 2008 conference this year. I chose to go for Wednesday's and Thursday's talks, but next time I plan to attend the whole conference. Overall it was well worth spending the time at the conference, meeting a variety of interesting people. I know people always say you learn more in the bar afterwards but I would say there would have to be some pretty intense knowledge exchanges to beat the information I picked up over the two days.

Value Delivery For Agile Environments [Tom Gilb]

I can sum up the talk in three words 'Measure Measure Measure'. Tom Gilb used his keynote to explain EVO, an envelope framework to surround a smaller development-centric process, which was in this case Agile. He sees Agile as deficient in that it is a development process geared for delivery, but less thought is put into what you actually deliver.

The problem comes then with what do you measure, how you measure, and then how do you interpret those metrics. By doing this and combining it with a fast deliverable methodology like Agile then you end up with constant iterations with feedback able to deal with the changing nature of the world (most probably defined by requirements).

I felt that the talk had an implicit feeling of 'How To Survive'. You need to identify your stakeholders, the people that determine the success and failure of your project and make sure that the needs of the most important and influential ones are met. If they like what you are doing by meeting and possibly exceeding their needs then you are more likely to gain extra resourcing as you are then seen as a successful group.

Bits And Mortar [Ric Parkin]

Like an extended episode of *Grand Designs* we were taken through an analysis of buildings. Well, no not really, but some of the theories that we are using in computer science have been looked at before and not only recently, in a completely different problem domain, and this field is architecture and the evolution of buildings.

I'm not the world's biggest fan of analogies because pedants always want to poke holes in it and take it off course thus negating any benefit from using it. I wish I never used analogies but I am like a lemming following everyone else. Luckily there was a thoughtful audience and the core was suitably abstract to avoid those problems.

The basis of the talk was the work of architect Christopher Alexander. He posited the theory of patterns which obviously directly relates to what engineers are doing right now, and sometimes he doesn't even refer to architecture and buildings. Due to my ignorance all of this was completely new to me, and I could take a lot away from the talk because a lot of the ideas of the evolution of a building (and therefore design) is directly applicable to the realm of computer science. If you look at a building as a finished fixed product after it has been completed then you forget about the lifetime of the building and how it evolves, much like a codebase. Knowing when to rebuild or rip-down parts requires suitable knowledge of what you are doing and you can also apply patterns other people have proved to be successful subsequently.

Unfortunately for Ric the talk will be forever remembered as the place he uttered in public 'I don't mind introducing bugs'.

Practical Multi-Threading [Dietmar Kuehl]

This talk covered the basics of the new C++ standard. It was a packed room so a lot of people are interested in this area.

It certainly looks like writing multi-threaded applications will be much less code in C++ than it has traditionally been. I like getting more functionality for less code. Items like condition variables will be supported in the C++ Standard Library. There was also a brief part about some of the TR2 features (C++0x + 1), such as futures, which makes using concurrent processing of independent blocks of code even easier and simplifying the synchronisation. This will be helped even more by the lambda expressions, as I can see some simple operations can be kicked off and calculated independently in a single line.

There was some coverage of Intel's Threading Building Blocks which provides concurrent containers and concurrent algorithms like `parallel_for` or `parallel_reduce`. This all provides some higher level semantics for expressing the concepts of multiple threaded processing.

When Good Architectures Go Bad [Mark Dalgarno]

This was more of an interactive session where people's experiences fed directly into the talk, so it means each time you would hear a write-up about it there would be a different opinion. Luckily my group had some interesting anecdotes. I do wonder why we all stay working in computers if we suffer this much abuse!

We used our experiences of the world to come up with examples of where the architecture had begun to 'smell' and what this represented. Looking at case studies we attempted to identify and find potential solutions to eliminate these smells. For my example of a system that had been going for a very long time through so many different platforms, teams, languages, I said to cancel it because it was not making enough money to warrant its existence. The most frightening solution to architectural decay, which also came up at the SPA conference, was to 'Kill The Architect'. I thought I was cynical.

This was a talk where you got more out of it if you put more into it. Hopefully Mark will put up some of the responses he got from the audience on his blog [or later in this very article – Ed.]. In fact, he could probably write a very frightening book about it.

The Future Of Concurrency In C++ [Anthony Williams]

Anthony is the maintainer of the Boost.Threads library. He went through some of the more complex parts of the upcoming C++0x and C++0x TR2, as well as what is available through Boost.Threads now.

One of my favourite parts of the entire thing is the concept of thread-local storage as a built-in keyword. No more `GetTLS` and the suchlike. I could immediately see a use of a static member of a class that is per-thread in order to create a memory allocator for STL containers which would allocate via only the thread's heap. If you know that some information is local to a single thread then you won't have any memory contention (in the program – I am not thinking about the hardware or underlying implementation) to slow down the memory access. You have to have a clear design and use of this though otherwise you could blow your program's brains out, but also that design works very nicely with thread pools...

Unfortunately some of the higher level concepts will take until probably the next standard TR2 to get to compilers. Of note are thread pools and futures. Futures mean you can run a thread for a calculation, start it off in a single statement, check the result after doing some more work, and it will wait until the result is posted. The result will then propagate any exceptions that had occurred on the calculation thread.

A large portion of the C++0x threading implementation is available through Boost.Threads thanks to Anthony's sterling efforts for 1.35, so you can already have a play.

Adobe Source Libraries : Overview And Philosophy [Sean Parent]

I can think of worse things to be remembered for, but I hope that Sean Parent is not only thought of as Alexander Stepanov's boss. He heads up the Software Technologies Lab at Adobe that creates generic libraries which are used in all Adobe's products. This talk was divided into two sections, one concentrating on the data structures and generic programming, and the other about declarative UI.

A very interesting part of the talk was the way he said he was using Alexander Stepanov's skills, basically 'Write a book defining generic programming'. The research for this has lead to lots of leaps forward for Adobe's programming technologies.

This talk started a little above my head by defining Regular Types which are very similar to the definitions and rules that can be derived for functional programming except you can have side-effects. This then provides the basis for generic programming.

We had a close look at the ideas behind the Move library currently maintained by Adobe but could go (back) into Boost. This library uses Return Value Optimisation to minimise and eliminate copies, and I was surprised to learn this was through the use of passing by value rather than by reference.

There was also a look at the `copy_on_write` functionality which means on object is only copied when it is written to. This provides the platform for Adobe's history tool in Photoshop and minimises memory impact. There was a look at the Forest container which approaches the binary tree in a very tidy way. They also have a string library that uses the move library so concatenations are much more efficient.

They are the right tools to solve certain datatype problems in a very concise and efficient fashion. The idea behind it all is to only use small pieces of code as building blocks towards the larger solutions. What they want to do is reduce the number of lines of code defining their applications by a very large factor (like from 3 million to 30,000).

The second half of the talk was based around declarative UI and the structures Adobe have put in place to solve problems that still exist to this day. I've looked at the two main libraries involved, Adam and Eve, before. Both really amount to being constraint solvers, one solving the data and one solving the layout.

The layout library is probably the simplest to explain as it works out, from the size algorithms you provide, the layout that follows the guides that have been set up. This also means that it can scale the layouts with relative ease. The property library is used to solve data dependencies (a lot of which are typically cyclic) for user interfaces.

The papers and documentation on the Adobe site can probably explain all this much better then I ever could, but they are all interesting building blocks for solutions to some overlooked problems.

Overall I enjoyed the talk, particularly the first half as their practical approach to implementing generic programming with real benefits, was quite eye opening. Unfortunately the talk was really badly attended as both halves were up against some tough competition (especially about functional programming), but hey they all missed out on some really good stuff.

Olve Maudal <oma@pvv.org>

Just back from 7 days in Oxford attending the ACCU conference. Around 300 delegates, interesting program, very suitable conference location and excellent organising comittee lead by Giovanni Asproni. The conference was packed with people that really care about programming like myself and they all behaved as if we were long time friends. It was this feeling of… feeling of… coming home. The conference was a superb experience. I just wish I knew about this conference years ago and I will definitely try to go next year as well.

On Tuesday (Day 0) I attended a one-day tutorial about Erlang – the programming language. I do believe that we are about to see a paradigm shift in the way we think about programming computers. Declarative and functional languages might soon play an important role, also in the industry. The tutorial was presented by Joe Armstrong, Mr Erlang himself. I enjoyed the tutorial very much. I hope to get more time to do play around with Erlang soon.

Tuesday night a group of us went out for a curry at Chutneys. Highlight: The bill. Why? As someone proudly announced: 'Hey folks. This is fantastic! We are 23 geeks and the bill is exactly 529 pounds!'. When everybody around you smiles and finds that amusing – then you know you are among friends.

Day one

Wednesday started with a keynote by Tom Gilb. Tom was concerned about delivering real business value to all stakeholders in a project. He proposed to add an 'agile envelope' around the agile methodology and lean principles. Several in the audience were provoked when Tom insinuated that agile practitioners are not really trying to deliver business value… and you know that you are at the right conference when someone just stands up in anger and shouts (something like) 'Tom, what you are saying is wrong!'. He continues to promote the Impact Estimation Table, and, as usual, his solutions involves measuring and quantifying stuff – which makes me sceptical.

Roger Orr talked about writing programs in a networked world. I found the talk interesting even if it was not too much new for me here. Having worked with networked applications for over a decade, most of the stuff was known. At the same time, it is always useful to get reassurance on things that you think you know. Key messages:

- Good networking interfaces are the key to good support and maintenance.
- Prefer higher level abstractions, allowing for multiple potential transport protocols.
- Make sure that you handle versioning issues.
- Security usually conflicts with other goals (eg, supportability) and it is often not possible to add proper security late to a product.

How to become Agile was a talk by Jutta Eckstein about introducing agility to a project or to an organization. The key idea was that a successful transition is impossible without involving management. But at the same time, introducing agility top-down does not work – trust is lost at the beginning. People are often looking for recipes, which is kind of opposite to what agile methods is all about. It is useful to identify and empower change agents that can assist the process. The change agent is often someone from inside, but they might need some support from outside at first. People tend to listen more to external people than internal people even if the message is the same. As most agile experts seem to agree upon, Jutta claimed that doing retrospectives is the most important agile technique. It is a big mistake to skip the retrospective sessions. Between projects you might need to use a whole day, while an hour might be sufficient between iterations.

Perhaps the most entertaining talk at the conference was Robot Wars by Ed Sykes and Jan-Klaas Kollhof. They believe that there is an exciting robotics market about to evolve, and they have been looking into different development tools. They demonstrated how to use Microsoft Robotics Studio to program Lego NXT robots. Then we got a demo of free alternatives for robot programming. For their final demonstration, Ed and Jan-Klaas added a camera purchased from mindsensors.com to show how a robot could chase another using a very simple application. I have an NXT myself and I thought this was really great stuff.

Day two

There is something in the air… said Simon Peyton-Jones, Mr Haskell himself, during his opening keynote on Thursday. His talk was about 'Caging the Effects Monster' – the next decade's big challenge. The key message was that in order to improve and reach nirvana in programming we need to be able to control the effects and implement large parts of our programs without any side-effects. Simon demonstrated some really nice examples and rationale for functional programming. He also presented

Kevlin Henney entertains a packed room

strong indicators showing that functional languages are attracting substantially more attention these days.

'The Selfish Object' was a talk by Kevlin Henney. The key idea was that instead of focusing on what an object can use or be given, you should focus on what it wants. Need and want is not the same thing. The same goes for object interfaces. When extracting interfaces, focus on the usage and not on the implementation, for example do not name the interface after the implementation, but find a name based on client usage. Avoid singletons, there is never a real need for them, you can always parameterise from above (PfA) instead. Don't use approaches like Template Method (NVI). Through techniques for controlling dependencies, such as PfA, dependency inversion, role-based naming, and more, you might end up with a better radial architecture (onion ring) than the more traditional one-way architecture (layer cake). Later same day, John Lakos summarized nicely: 'I want to depend on the interface, the whole interface, and nothing but the interface. So help me Kevlin.'

Just after lunch there was a BoF session about local ACCU groups. Since I am involved in a lot of geek activities in the Oslo area, I was interested to hear what kind of things local ACCU groups were discussing. How to get speakers and how to attract people to the events was discussed. Apparently getting a location for events is difficult. In Oslo we often use pubs for small events (up to 120 people), this seems to be more problematic in the UK for some reason (perhaps there are not that many pubs in the UK?). The ACCU group in London have had success with borrowing meeting rooms from the big banks. Another group had managed to use the computer section of a book store, a really nice idea. Recording of presentations was also discussed. But, first of all recording introduces a lot of work and also the speaker and audience is less likely to interact and engage in interesting discussions if recorded. This aspect is also true for the ACCU conference. Recording the sessions removes the magic.

John Lakos considers taking on his entire audience.

John Lakos ran through 562 slides in his talk: 'Toward a Common Intuition and Reusable Testing Methodology'. It was an excellent presentation, but it was deep stuff and I have to admit that was not able to absorb all the ideas – it was like drinking from a fire hose. But there was a very interesting and solid discussion about what it means to be 'the same' and what the salient attributes of something are. In C++ a lot of errors arises due to ignorance to these subjects. John, what about dividing this talk into two, where the first is named 'The same? What the hell do you mean?', that would be more like sipping a superior single malt.

The next session I attended was about 'Memory Allocation' by Andrei Alexandrescu. The main message was: If you try to write your own allocator, you will fail. Over and over again, we see that the best general purpose memory allocator outperforms a special purpose memory allocator. If you have identified, through proper profiling, that you indeed have some specific needs, then you should use a reaps allocator (regions with free-list), otherwise go for the Doug Lea memory allocator.

The last session of Thursday was a special version of 'Just a Minute' hosted by Ewan Milne. Funny, but perhaps not so useful. As I did not know about the 'Just a Minute' concept, I thought (for some reason) that we would get some sort of Lightning talk session. We have used this format (1-10 minutes talks) successfully at several events in Oslo. I would love to see a session of lightning talks at ACCU next year.

Day three

The opening keynote on Friday was presented by Andrei Alexandrescu. He talked about fundamental challanges in programming languages, and briefly introduced Stepanov's litmus test: If you can't implement max, swap or linear search properly, what are your chances to implement really complex stuff? To kind of underline the point, Andrei demonstrated an even more fundamental problem, even implementing the `identity()` function is really complex in C++. Fortunately, with C++0x we are apparently moving in the right direction.

Despite being a dedicated Emacs power user, I have to admit that I sometimes envy the tools that Java developers have available – Eclipse being one of them. I wish I could use Eclipse on C++ code as well. For the last few years I have been downloading the latest version of Eclipse CDT once in a while to give it a go. For now, I do not see that it adds any value to my C++ development environment, but I am still optimistic because I can see improvements every time. When I saw that Peter Sommerlad was giving a talk about C++ Refactoring and TDD with Eclipse CDT I thought I might learn more about the state of CDT – and I did. Some of the new refactoring tools that have been added look interesting, but I got the impression that they are still quite fragile for variable C++ coding styles.

Testing is a way of showing that you care. This was a key message from Kevlin Henney in his talk 'Know Your Units'. There are many testing techniques – unit testing being only one of them. But it is important that you distinguish between what is a unit test and what is not. A test is not a unit test if it uses external resources or if it require a particular order of execution. By focusing on doing unit testing correctly, you will often be forced into making sound design and architectural decisions. Tests that are not unit tests according to the definition might also be very useful but they serve another purpose – often they focus on finding bugs. A useful technique when writing unit tests is to prefix the test name with 'require' as in 'require_that_sqrt_of_4_is_2()' rather than 'test_sqrt_4_is_2()'. Your tests should look like requirements and this naming style will guide you into writing better unit tests. Other guidelines:

- The more general a method gets, the less useful it is for a particular application.
- Get rid of your singletons, they make your code untestable.
- Don't ever invite a vampire into your house, you silly boy – it renders you powerless.

Before the Speakers Dinner, two teams met at the squash court to settle the long term debate of whether braces should be aligned:

```
if (is_ready())
{
  do_foo();
  do_bar();
}
```

or disaligned:

```
if (is_ready()) {
  do_foo();
  do_bar();
}
```

was (naturally) playing for the 'disaligned' team and it was a fierce competition for about an hour and a half before we gave up… failing to declare a winner. It was decided to bring in more combatants and do a boat race later in the evening to settle the debate. If I remember correctly the 'disaligned style' team lost, but as you know, in a boat race having most supporters and the biggest team is not an advantage…

Seb Rose celebrates another victory over Jez.

Day four

I would have liked to see Roger Orr presenting the keynote on Saturday morning, especially since I later was told that it was a really good one. But I did not get to bed before 5am Saturday morning and sometimes you have to prioritize hard.

Detlef Vollman gave a talk about 'C++ for Embedded Systems'. This was a particularly useful session for me since it is exactly what I do for a living. Some messages:

- In embedded systems, power consumption is often the biggest problem.
- Immutable strings make sense, so you might need to implement your own string class.
- In low-level classes you should not use dynamic memory allocation.
- Don't fall into the OOAD trap where you only analyse a system top-down, for embedded systems you must also use a bottom-up Lego approach.
- C++ is a multi-paradigm language, which is very useful for embedded systems.
- Only use OO if it really gives you some benefit.

For the last year I have been following the C++0x process closely, so felt I had to attend C++ 2009 in 90 minutes by Alisdair Meredith. I already heard about most of the things that are going into the new standard, but my knowledge is superficial. For example, when I first saw the proposal about `rvalue` references I realized how useful they are for making dead hard quiz questions, but after Alisdair's talk I understand more about why some of these things are important additions to the language.

Finally, there was a panel debate about the 'State of the Practice' lead by Giovanni with Tom Gilb, Hubert Matthews, Russel Winder, Peter Sommerlad and James Coplien in the panel. They all seem to agree that as an industry we have really screwed up badly. Sure there were a lot of good points made, but I suspect they have a somewhat biased experience base. Big names like these guys are often brought in to fix stuff in failing projects rather than watching successful projects completing a masterpiece. In addition, it is always comfortable to be the one criticizing instead of being optimistic – being a pessimist is the safe bet in all things with a large degree of uncertainty. But at the same time, the session was indeed interesting. Some stuff that was discussed:

- Are there too many lay programmers out there?
- Do we need to become a registered profession?
- Perhaps we must split CS into displines like telecoms, banking, military, and so on?
- Are we going into a cultural rot?
- Is software development a normative discipline?
- Do customer buy a service or a product?
- Does better compilers make it just easier for lay people to write bad software?
- Will making a registered profession have impact on free and open source software development?
- Is the free market for software working?

And then the conference was over.

Sunday morning. 6am. Three alarms go off. Must not be late for my flight back to Oslo. Looking out of window. England covered in snow. Wow! Prepared for a really bad trip home. Luck. Flight was just a few hours delayed. Wife and two kids. It's always nice to come home…

Mark Dalgarno <mark@software-acumen.com>

Value Delivery for Agile environments [Tom Gilb]

Tom Gilb kicked-off the ACCU conference on Wednesday with a controversial keynote, for some, arguing that Agile methods do not have enough focus on delivering value to the people who pay the wages.

Tom began by claiming that agile methods are too self-centred in process and methods. This makes them good for focusing on programming tasks but they need to be supplemented with other methods in order to manage value delivery.

Perhaps unsurprisingly Tom suggested his own EVO method as the ideal method for doing this, although he noted that you could also use DSDM or RUP – both of which look more closely at value than Agile methods. Tom cited several recent examples of work done by Ryan Shriver of Dominion Digital on wrapping SCRUM with an EVO envelope. Ryan's findings provide evidence that this combined approach provides both a method for managing value delivery and a method for managing programmers.

A fundamental problem with XP and SCRUM (for example) is that they don't provide guidance on the highest-value/lowest-cost work packets. They have no alignment with higher-level business goals, argued Tom, and consequently no measurement can be performed with respect to business goals. This provoked some audience heckling but a full debate couldn't be carried on in the keynote.

Tom proposed the following framework for making smart decisions:

- Measure progress towards goals – burn rate (stories) isn't a good metric
- Get a better understanding of time, budget, people
- Are we using these in smart ways?
- How good are we at exploiting gathered information on the next iteration?
- Analyse this frequent feedback and adapt processes to correct and improve

Agile methods are very feature-driven – but according to Tom this doesn't help us work in terms of business goals – so there is a problem in deciding how to maximize the value to business. The key solution to this is to quantify business, technical, and organisational values.

Tom suggested that the main take-away from his talk was the use of impact estimation tables to measure value delivery/costs. These give a clear understanding of options. There's more on these in his book Competitive Engineering: A Handbook For Systems Engineering, Requirements Engineering, and Software Engineering Using Planguage.

Using Evo concepts – budgets, goals, design ideas, impact evaluation, requirements spec. with Planguage, estimation, planning and tracking estimation to wrap Scrum concepts gives a complete environment for delivering value to stakeholders – not just the customer. Agile methods are too-focused on the customer. As an example medium-sized projects have 40 stakeholders – 50% internal, 50% external. They are stakeholders because they have requirements and need value to be met. Another take away point– are you identifying all important stakeholders in your projects? Someone has to analyse values and needs of all stakeholders. If critical stakeholders are denied what they want they can probably destroy project.

Systems are all about building potential to realise stakeholder value – if a system isn't used then no value can be generated.

As an aside, Tom took a straw poll of use of agile methods in the audience – Scrum was much more popular than XP – suggesting that XP is on the way out?

Tom described 10 principles to help get business value from projects.

1. Critical stakeholders determine the value.
2. Values can and must be quantified. (Useful to get to actual values e.g. robustness that can be agreed. Use as clear targets for architecture, design etc.)
3. Values are supported by Value Architecture. (Architecture design is an optimisation problem – most architectural decisions impact multiple values.)
4. Value levels are determined by timing, architecture effect and resources.
5. Required value levels can differ for different scopes (where, who). – (Different stakeholders have different needs, these also change over time.)
6. Value can be delivered early. (Intentionally target the highest priority stakeholders and their highest priority value area – deliver them value early and continuously.)
7. Value can be locked in incrementally. (Give the system to real users – they won't give it back if they benefit.)
8. New values can be discovered. (Expect to discover new stakeholders and new stakeholder values. Developers must be in dialogue with stakeholders/users. This partly arises because stakeholders will come to believe that you can help.)
9. Values can be evaluated as a function. (Tracking stories and burn rates is the only feedback you get in Agile methods. But productivity is defined as reaching goal levels of organisation.)
10. Value delivery will attract resources. (If you are good at delivering value you can expect more funding. Managers like to be credited with success.)

Higher adaptability in a system/organisation is an investment viewed as something that returns value over a longer-term period – Tom argued that this long-term view totally absent from agile. There are however some responsible corporations e.g. HP – in that in those environments you can do this sort of long-term thing without too much debate.

Often nobody has the responsibility that the value be realised – Tom proposes a Chief Value Officer to have this responsibility. Unfortunately he didn't have enough time to go into this in more detail…

Snowflakes and Architecture - Layers considered harmful [Steve Love]

Steve Love's session at the ACCU Conference was billed as taking a 'suspicious look at the traditional layered architecture, and suggest[ing] some ways it can be improved upon, resulting in an 'architecture' that resembles a snowflake more than it does a cake'.

Steve was straight in with the boot beginning by noting that the main difficulty with layer diagrams is that the diagrams are often only what you get.

Layers are created to separate concerns but often business logic leaks into different layers – this leads to untestable code due to unintended dependencies between layers e.g. the business application that must be tested with a GUI. Another problem is that sometimes a layer is there only to pass data through to lower layers.

Steve's talk described a different, more granular, approach to making a more adaptable architecture by design and looked at the mutual influence of architecture and design.

Robert Martin notes that bad design is rigid, fragile (due to knock-on effects of local changes) and immobile (can't move things around for reuse, can't disentangle software). Booch suggests that instead of layering, we arrange applications into smaller grained components that communicate together. So good design is cohesive, decoupled and layered, Steve questioned whether these qualities are enough…

What a user sees as quality is different from what developer sees as quality. His claim is that simplicity in software architecture is key, is easy to measure, but is not straightforward to achieve.

The conference prepares to settle one of today's thorny programming problems while Tom Gilb considers the value proposition.

To illustrate what he meant Steve introduced concept of Dependency Horizon, which is the number of steps you need to bring in for a particular dependency (think of it as the number of steps between components, packages or modules if you will). A far dependency horizon can become hard to manage and can increase chance of circular dependencies.

Decoupling, e.g. through intermediate objects or interfaces, improves maintainability. This can be done by identifying abstractions and pulling out pure interfaces. This shortens the dependency horizon and so maintainability is improved.

A key concept here is the Dependency Inversion Principle

- High level modules should not depend upon low level modules. Both should depend upon abstractions.
- Abstractions should not depend upon details. Details should depend upon abstractions.
- This allows abstractions to depend on each other, but not on their implementations.
- The Dependency Inversion Principle is layering in the small.

The Singleton pattern came in for a lot of bashing in Steve's talk. Singleton is the counter-example of DIP – it is the antithesis of detail depending on abstraction. The pattern has lots of problems – hard-wired dependency, dependency from within, testing is difficult, rigid/fragile/immobile, unpredictable ownership, unpredictable lifetime, difficult to handle when multiple threads are present. Conclusion – Singleton pretty much violates all of Martin's design principles.

Steve's answer is Unsingleton – code should work with what it wants, but don't let it take it for themselves. What client code doesn't know about the implementation of a service can't hurt it. Steve's solution to many of the problems was to Design to an Interface. This can be realised in different ways in different technologies:

- Interface keyword in some languages
- COM/CORBA - IDL
- C++ pure virtual base class
- Duck-Typing (ducking the whole idea of typing) - C++ Templates, Ruby, Python
- C# and Java Generics

The key thing about interfaces is their substitutability.

- Inheritance is the tightest form of coupling possible. Inheritance – polymorphism by dynamic despatch and virtual functions.
- Genericity – polymorphism by generics and duck typing – Containers, Iterators and algorithms, Traits and Policy classes.
- Overload – polymorphism by overloaded functions – member function overloading, global functions and operators. Can be particularly powerful in creating interfaces.
- Coercion – polymorphism by conversion, implicit casting, constness.

Testability is an essential property.

- You must be able to Test independent parts independently

- Interfaces give substitutability – if software under test depends on database etc. then change it so that it must depend on interfaces to the database, this lets you mock or replace the database
- Substitutability underpins Mockability. Gives you ability to mock out services that might only available on target device (e.g. a pda or phone) and do debugging on the development PC.
- Design to an interface.

(A quick aside: When I first started programming I didn't think about this very much – we weren't taught it and nobody around me considered it as a property that could be designed in. As I read more about software development I began to take more notice of this and began to look for testability in design documents. Debuggability is a related concept that supports testability e.g. I built a logging add-on for some quite complex database code I inherited and this really let me get to the bottom of some quite complex client-server interaction problems.)

Parallel Development.

- Working to interfaces enables parallel development, continuous integration, testing
- Also supports outsourced development
- Adaptability – when using interfaces plugging in a new component just like an existing one is trivial.

I did have a question in my mind about how much effort and knowledge is required in order to develop these interfaces. Does one need to have written the interfaces a few times in order to get them right?

Flexibility, Generality and Reuse

- The false idols of OO?
- Usually done with inheritance but these led to big class hierarchies which were unusable rather than reusable.
- Interfaces provide the means of reuse. - component architecture provides the means of flexibility – make stuff talk to a wire feed rather than the UI say
- Generality – can be reused in different context

Fat interfaces are less useful than small interfaces since they bring in unwanted dependencies. They must be designed according to what client code wants, not what it can use. Small, specific interfaces allow better reuse. Parameterize-from-above is also part of the solution. 'Don't call us, we'll call you.'

Alistair Cockburn propose Hexagonal architectures – promoting the idea of the application as a service, with ports and adapters. A port defines the contract for adapters – this leads to pluggability. Making adapters with their own ports leads to software as a collaborating set of components. Steve notes it's an attractive design but that it requires discipline and can lead to circular dependencies. At the back of his mind he did seem to be worried about the danger of replacing spaghetti code with spaghetti interfaces. One suggestion was to try and organise interfaces into layers to help. The key here he suggested is to break layering down into its constituent interfaces. We would still layer components but we don't make them depend on being in a layer. This allows decoupling within layers.

So – what about the snowflakes? Look for Steve's slides on the conference web site and you'll get the picture (pun intended).

When Good Architecture Goes Bad [Mark Dalgarno]

I'm writing this two weeks to the day since I stood up in front of just over 30 people to lead my session 'When Good Architecture Goes Bad'.

My plan was to present some examples of architectural decay, to collect some examples from the participants and to explore how things could be improved. I was particularly interested in the value and cost of work done to prevent architectural decay. It seems that developers agree that preventing such decay is a good thing but I was hoping to collect some examples that could make the financial side of things a little clearer as it seems to me that there is a disconnect between developers and managers in this area. It wasn't too clear from the session description in the programme that this would be a workshop so first up I offered anyone who just wanted to sit and listen the chance to leave the room – there were no takers…

Richard Harris BSc, MSc ... PhD writes

Richard Harris's presentation was a masterpiece of understated genius. Subtly made and yet profound, this modest intellectual leviathan questioned our assumptions of the very nature of the universe. In a dazzling 90 minutes, this renowned mathologist exploited his supernatural mastery of PowerPoint to transform complex mathematical concepts into simple visualisations that even we mere mortals could follow. No more shall we curse the manufacturers of headphones, knowing as we now do that their products are doomed to tangle by the fundamental laws of reality.

Oh, and there were some folks talking about C++.

After a few introductory slides the first exercise asked participants to discuss examples of architectural decay from their real-life experience. I collected these on a flipchart:

Examples of architectural decay

- A single class used as a dumping ground
- Cancerous wart – ever increasing coupling between modules, packages etc.
- The number of programming languages used on the project increasing over time
- New interfaces added over time, but old interfaces still maintained (and never deprecated)
- Lots of code clones (copies and near-identical copies)

I then presented some examples of architectural smells (problems in package/class/subsystem/layer relationships, overgeneralization, etc.) and whiffs. Whiffs are subtle smells – no one on the team can tell you what the intended architecture is, time to implement changes increases, etc. The second exercise asked the group to come up with their own examples:

Architectural smells associated with these (and other) examples:

- Mismatch between documentation and software, mismatch between comments and code
- No clear responsibility for the architecture
- Implementation = specification

Use of a proprietary language compiler

- Pile of s**t from the start of the project
- Clone and own as the main way of doing reuse
- Insufficient decomposition
- Knowledge of architecture held by a decreasing number of people

The bulk of the session was taken up by two case studies. In the first case study an outsourced project had run into problems over a period of years. The company detected a considerable decrease in productivity over this time and the participants were asked to decide whether architectural decay could have caused the decrease in productivity and what they thought of the company's proposed solution.

Participants' observations included:

- Communication needed to maintain architecture
- Management needed to maintain architecture
- Someone required to shepherd the new team before they can get going
- Insufficient knowledge transfer process when software first outsourced.
- Team selection is important when assigning new roles.
- Unclear when productivity decline happened. Why didn't company pick it up sooner?
- Who owned the software & the architecture – tests.

A second case study looked at a situation where a software system had been developed in three separate sites but the company had just closed two of

the sites. Participants were asked whether moving maintenance to a single team at one site would cause the software architecture to decay.

This time participants' observations included:

- Fewer people, resourcing could contribute to decay
- Knowledge transfer/must learn new bits – again could cause problems
- Subtle differences between architectures of the different parts could cause problems – initially it's a comprehension task
- Domain expertise was lost when two sites closed – could indicate significant loss of architectural knowledge
- What were the future plans – is there an implied refactoring?
- This situation needs management to go into the project with open eyes – non one was convinced this was the case
- There might be an urge to change the architecture which could be risky
- Not-invented here/cultural differences could lead to problems
- The motivation of the new team was questioned, skills drain could occur
- What was the background motivation for the change? This could set the tone for future work.

If you would like to try these yourself, the case studies are available at: http://www.software-acumen.com/articles-and-essays/

I then asked participants to travel back in time and come up with some ideas for maintaining architectural integrity in their previously noted problem projects:

- Encourage people not to do clone and own – but how to do this?
- Change the development process (again how viable is this, would management back it?)
- Embed a culture of refactoring (again but what if management won't allow it?) (and note – some developers may want too much refactoring, be too keen to rewrite)
- Visualize the technical debt – detect architectural decay using tools
- Make responsibilities clearer.
- Add automated (architectural) checks.
- Cancel the project (earlier).
- Rewrite (earlier)
- Kill the architect (as noted by SPA 2008 participants also)
- Spread architectural knowledge.
- Spread the architectural work.
- Have frequent communication between whole group – up to three times a day.

The session covered some of the same ground as Tom Gilb's keynote 'Agile Methods Lack result management', Steve Love's session on 'Snowflakes as architecture' and (it later emerged) Dirk Haun's session 'Rewriting not recommended'.

Michael Foord <fuzzyman@voidspace.org.uk>

I'm writing this having just returned from the ACCU Conference in Oxford. Last year Mark Shuttleworth was one of the keynote speakers, and the year before that the eminent Guido Van Rossum – so after my talk was accepted I was expecting quite an academic and 'high powered' conference (in other words I was very nervous about speaking there and didn't know what to expect).

Of course in reality it turns out that although they do have some very good speakers, it really is a community organised event with some fun and down to earth people. One thing that was a real challenge to my mindset was to spend a bit of time with a genuinely intelligent person (Dirk Griffioen) who chooses to program in C++.

First things first, my talk on IronPython and Dynamic Languages on .NET went very well. I had a good audience (around fifty people I guesstimate) who were very responsive and asked a lot of questions. About half were .NET programmers and half Python programmers. They seemed to like Resolver One, our Python programming spreadsheet, and were impressed by first class functions and decorators in Python.

My favourite part of the talk was my deliberately provocative statement on static typing:

In statically typed languages, it turns out that a significant proportion of language 'infrastructure' (boiler-plate) is there for the sake of the compiler rather than the programmer.

In C# this includes delegates, interfaces, generics and type declarations which are all obsoleted by a dynamic type system.

Fortunately this was taken with a smile by most people there.

The Thursday keynote was 'Caging the Effects Monster', on controlling side effects in programming, by Simon Peyton-Jones, the creator of the GHC Haskell Compiler. He wasn't just advocating pure functional programming languages, but was encouraging developers to change their programming habits. He did a great job of explaining how to do this, but whilst he mentioned parallelism I didn't feel he explained why very clearly.

This was followed by Joe Armstrong talking about Erlang and then Simon doing a three hour tutorial session on Haskell – I spent the whole day learning about functional programming! Simon's examples were using Haskell for shell scripting (!) and the xmonad window manager – he was very much touting Haskell as a general purpose programming language. He demonstrated using Monads for IO and was easy to follow, although by the end of three hours my brain was starting to hurt. I have, though, promoted Haskell higher up the list of languages I would like to learn.

One of Simon's early examples in his tutorial was writing a Haskell function to traverse a graph of atoms to find neighbours (the 'n-shell'). The function was fantastically simple, but was exponential. Simon commented that this could be solved by using memoize. Inspired by this, in my talk I showed how easy it was to write a memoize decorator in a few lines of Python.

Whilst chatting to Dirk, he tried to convince me that memory allocation in C++ is simple these days, and that most people who have had painful experiences of C++'s complexity are remembering an older C++ which is now much improved. Unfortunately the Friday keynote, by Andrei Alexandrescu (a C++ expert and a collaborator in the specification of version 2.0 of the D Programming Language), did much to convince me of the opposite. This was an hour and a half of examining, in detail, the terrible problems of trying to implement general purpose identity (lambda x: x in Python for all cases), min and max functions in both the current standard of C++ and C++ 0x (the forthcoming standard). This includes dealing with rvalues, lvalues, passing and returning arrays, consts and non const values. Even at the end of Andrei's presentation of the identity function – which he said took virtually a day to work out – someone in the audience pointed out a corner case it couldn't handle. What a lot of awful, terrible, unnecessary complexity.

I also got a chance to demonstrate Resolver One to Simon Peyton-Jones, who was particularly impressed with the fact that cell ranges are iterable. This means that you can have formulae like =SUM(val for val in A1:D8 if val > 10), or use the filter function with a lambda predicate and a cell range: =SUM(filter(lambda x -> x > 10, A1:D8)). He says he has been trying to get features like this into Excel for years. Yay for us!

Pete Goodliffe <pete@cthree.org>

Day one

Wednesday kicked off with a keynote by Tom Gilb. This was a provocative talk on his thoughts on software development process and his Evo methodology. He managed to tread on the toes of the agile contigent, and interestingly suggested using Evo as envelope around an agile process. One of his beautifully inflamatory statements was that 'agile programming

does not attempt to quantify the value of various pieces of work, so you are not able to pick the pieces of work that have the highest value, and so agile processes fail to deliver (as much) value'. Or something like that. I'm not sure I agree.

Ric Parkin's talk on software design walked us through Alexander's seminal architectural books and considered their applicability to software design. Not new ground, but very though provoking. Jez and I on the back row took this to the logical conclusion and came up with 'Grade 1 listed software' – the kind of thing that should not be touched without written planning consent.

Ric's most amusing quote, which will be repeated back to him many, many times in the future was 'I don't mind introducing bugs'. Thanks for that Ric. I can't dig him too much, though – he did give my book a free plug.

Day two of the ACCU 2008 conference... another barrage of technical information and geeky entertainment.

Thursday's timetable had a refreshing functional programming track, which was headlined by Simon Peyton-Jone's keynote: 'Caging the Effects Monster'. Great stuff. I can only admire the man for being a fellow bare foot presenter! Simon took us on an entertaining journey into functional programming and how it can be used to minise the risks of 'effects' (or rather, dangerous side-effects) in our software.

We saw how the 'useful but dangerous' languages are gaining more 'pure' functional capabilities and the 'useless but safe' pure functional languages are gaining 'side-effects' to get actually stuff done.

The functional programming track continued with Joe Armstrong (self-confessed quirky Englishman) explaining the motivation for Erlang, and finishing with an eleven minute introduction to the language syntax delivered in five minutes.

Favourite Armstrong quotes:

- The operating system is for the stuff they forgot to put in the programming language
- Designing code for fault tolerance is the same things as designing code to scale
- No one's ever done an MRI scan of the brain whilst you're writing a concurrent program
- I'll do the 11 minute introduction to Erlang in about 5 minutes, and then do a 1 minute encore
- Defensive programming is evil – you don't do any defensive programming in Erlang

Later the day included John Lakos on a heroic romp through 564 slides in 90 minutes whilist providing a classification model for objects, in order to aid testing, and to validate the new C++ scoped allocator model.

The conference sessions closed with a geeky version of 'Just a Minute'. Great fun, and I'm obliged to mention it mostly because not only was I on the panel, I won :-)

Day three

So is it Friday or Wednesday yet? We've been held captive in a zoo of programmers for far too long, and the toll is starting to show. We're all going slowly mad, or technical, or both.

Friday at ACCU 2008 was just as rammed full of tech as the previous days, with another full track of functional programming sessions nestling alongside the traditional C++ talks, as well as sessions on rewriting code, packaging with RPM, development process issues, and more, and more.

Highlights for me included Andei Alexandrescu's talk on grafting functional support on top of an imperative language – an excellent trip into the D language's core facilities that support programming in a functional

Just a programming minute!

style in the same codebase as imperative code. If you're even slightly interested by that concept, or by language design, I highly recommend you check his material out. Towards the end of Andrei's talk I was left disappointed by the design of invariant constructors which didn't seem anywhere near as neat and regular as the clever use of the D type system to enable the functional and imperative code to coexist and share state. It seems that the design is still in flux, and it'll be cool to see how it develops.

Kevlin Henney gave a typically amusing and insightful talk on software testing. An excellent Henney quote: 'In failure the software will reveal itself'. That is, when it goes wrong, you will learn about the structure and nature of a software system.

The day finished, and the night began (and – as ever – it was a looooong night) with the speakers dinner – another excellent ACCU tradition. That was rounded off with another new ACCU tradition, the boat race which solved once and for all which brace placement style is the One True Way – a score that the squash players earlier had spectacularly failed to settle. K&R lost, and so it has now been

```
void established::that()
{
  this->is_the_only(way, 2);
  do
  {
    it();
  }
```

And now we can all sleep at night. Except that they didn't shut the bar, and very few people did.

Day four

Saturday. More of the same – but with slightly fewer people attending on the Saturday it's a bit easier to get at the wireless network.

Many people are looking a little bleary eyed from last night's fun, and regaling us with tales of John Lakos' 41 one-armed push ups in the hotel bar at 4am.

Roger Orr's keynote on debugging showed us the seven deadly sins of debugging. and the seven (deadly?) corresponding virtues.

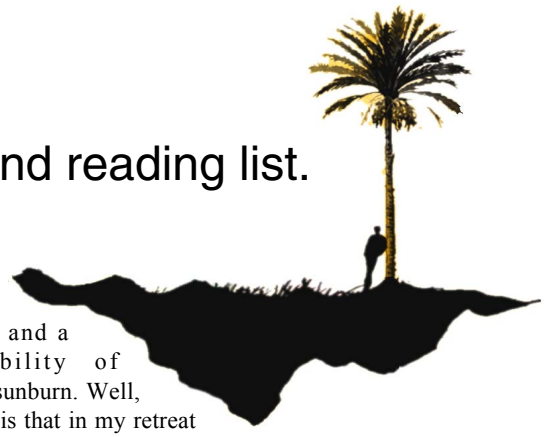
That's a flavour of the sessions, but so much of the conference takes place away from the PowerPoint projector. The conversations over coffee, dinner, and beer (that stretched very, very late into the night) are the high-point of the conference.

Jez (again)

There you have it – a view of the ACCU 2008 conference. Our thanks, as always, go to Giovanni and the conference committee, the speakers, to Julie and her team at Archer Yates, and, indeed, to the conference delegates. Hope to see you there next year!

Desert Island Books

Kevlin Henney shares his desert island reading list.



Kevlin Henney is no stranger to any regular on accu-general and probably no stranger to any ACCU member who attends the conference or reads CVu or Overload (and of course that's all of us, isn't it?).

If you asked him, Kevlin would tell you that he is an independent consultant and trainer with an interest in software development techniques and programming languages and that he has written two books. But of course there's more to the man than that. The main thing that stands out for me is that he always finds time to help everyone and to contribute.

It's obvious from speaking with Kevlin and reading his work that his interests do not only lie within programming. As he describes below, he also has a healthy interest in Science Fiction and reasonable taste in music. Although, I could never agree that *Wish You Were Here* was Pink Floyd's finest album.

Kevlin Henney

I guess there a quite a few of us who, as children, fantasised about being cast away on a desert island, living the life of Robinson Crusoe and other adventure stories. And now I'm being given the opportunity to revisit this dream with adult sensibilities and a geekish twist!

Alas, growing older is associated with some degree of self knowledge, a set of acquired habits and a whole load of other mental baggage, not all of which would prove useful in a castaway situation. For example, I know that I'm a complete town mouse. For me, the countryside is filled with strange animals, stranger people and lots of green stuff (although perhaps not as much as there used to be). It's where you go on holiday. It's where you pass through by car or by train to get to somewhere else. It's certainly more beautiful and pleasant than the banal surrealism of edge cities and suburbia, but it involves more separation from people than I'm normally comfortable with – I grew up in London and now live in Bristol (which is probably about as country as I can manage).

there are a number of books on programming languages and programming technique that are good candidates for enjoying in the shade of a palm tree

So what am I to make of being stranded on a desert island? Well, at least the weather will be better than living on this island. Sunshine! The gentle lapping of the waves on the beach! Living off nature's bounty! There's also the tropical storms, wild animals and poisonous plants to take into account, as well as the obvious lack of restaurants, convenience stores and 24-hour dial-up wild animal catchers. Oh, and a high probability of solipsism and sunburn. Well, the good news is that in my retreat from civilisation I should be reasonably well provided for. I can take five books and one or two albums. The flotsam and jetsam of the twenty-first century washing up on the beach will undoubtedly cater for a number of my town-mouse expectations (pollution, tinned food, Ikea furniture, etc.).

In terms of technical books, one I would definitely take is Grady Booch's *Object-Oriented Analysis and Design with Applications*, which I bought when it came out in 1993. To be precise, the second edition. The first edition, which preceded the second by a couple of years, had a focus that was slightly narrower as just *Object-Oriented Design with Applications*. The first edition was, for me, quite ground-breaking, but the second was more polished, comprehensive and definitive. Looking back at it, it is surprising how much of it still seems fresh and how much people could still learn from it, from little gems to deep insights. The book has a view of objects that is both sound in theory and robust in practice, using a mix of C++ code and Booch notation to make its points and walk through non-toy examples. In this book you will also find early discussion of design patterns and simple policy-based design, all ahead of the publications that later elaborated and popularised these ideas.

The first edition used a mix of languages, which is definitely a point for rather than against it, but the overall effect of the second book, reflecting a maturer, deeper and more integrated view of OO development, is enough to make it my favoured edition. Recently a third edition was published. It employs UML and a host of coauthors. I have a copy on my shelf, but I suspect it will probably never get read. Ahead of their time, the first two editions were of their time and reflected a clear and consistent approach. After such a long break, bringing in coauthors to enhance a work that reflected one person's world view is likely to result in something that is less than the sum of its parts, a potential disappointment for fans of the original. I had this experience with the original and recent editions of Glenford Myers' *Art of Software Testing*. Maybe the new OOAD book will be as good as the old ones. Maybe not. Either way, it is not the third edition that will be accompanying me to the desert island.

There are a number of OO and other '-ology' books that would have been good candidates for the trip. The Gang-of-Four's *Design Patterns*, Bertrand Meyer's *Object-Oriented Software Construction* (first edition) and Michael Jackson's *Software Requirements & Specifications* (not exactly a thriller (Paul: Pun intended?) of a title, but the nature of the book is better communicated in its subtitle, a lexicon of principles, practices and prejudices) have all moved my thinking on in some significant way. Ultimately, however, it's Booch that first got me interested in reflecting on my problem and solutions thinking above the level of the code, much to the benefit of my code. I first stumbled across OO in a book on Ada by Booch. At the time I was doing Fortran (actually, it was long enough ago that it was definitely FORTRAN) and the more considered and expressive approach to organising code offered by OO thinking and, specifically, his

What's it all about?

Desert Island Disks is one of BBC Radio 4's most popular and enduring programmes:

http://www.bbc.co.uk/radio4/factual/desertislanddiscs.shtml

The format is simple: each week a guest is invited to choose the eight records they would take with them to a desert island.

I've been thinking for a while that it would be entertaining to get ACCU members to choose their Desert Island Books. The format will be slightly different from the Radio 4 show. Members will choose about 5 books, one of which must be a novel, and up to two albums. The programming books must have made a big impact on their programming life or be ones that they would take to a desert island. The inclusion of a novel and a couple of albums will also help us to learn a little more about the person. The ACCU has some amazing personalities and I'm sure we only scratch the surface most of the time.

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

articulation of it was a complete revelation. This earlier inspiration probably influences my preference for his OOAD book.

Focusing on code, there are a number of books on programming languages and programming technique that are good candidates for enjoying in the shade of a palm tree. Looking across my bookshelf there are perhaps too many possibilities. The *C Programming Language* by Brian Kernighan and Dennis Ritchie (both old and new testament) still stands as a programming and language classic. Likewise, Alfred Aho, Brian Kernighan and Peter Weinberger's *Awk Programming Language* is a little gem of a book, often overlooked. Books such as Brian Kernighan and Rob Pike's *Practice of Programming*, Andy Hunt and Dave Thomas's *Pragmatic Programmer*, Jon Bentley's *Programming Pearls* (both first and second edition), Kent Beck's *Smalltalk Best Practice Patterns* and Martin Fowler's *Refactoring* are well-written, exemplary exponents of technique and, importantly for the castaway, each one is a jolly good read. But the ticket to the island probably has to go to *The Elements of Programming Style* by Brian Kernighan and P J Plauger.

As an aside, looking over that list of candidates and the chosen book, I reckon that I ought to be able to blag honorary membership of the Brian W Kernighan fan club. There does seem to be a deeper pattern here: Jez took a copy of Kernighan and Plauger's *Software Tools* off to his desert island in the last issue. (There's a thought, I wonder if it's actually the same island I'm on, just running in a different VM instance or time slice? Maybe it would be possible to communicate by opening up some kind of port or pipe? A bamboo one, probably. Or perhaps having such thoughts suggests that I've been drinking just a little bit too much sea water?)

Anyway, what makes the chosen book (as opposed to Brian Kernighan) so special? Before all the other books I mentioned came along, there was *The Elements of Programming Style*. Ideas that I have recently seen branded by some as fashionable are all in here, clearly motivated and clearly described: sparing use of comments; avoiding temporary variables; refactoring common code into functions; testing a program in small units; and so on. This book dates from the 1970s, which is not a decade known for its fashion. The example code is in PL/1 and Fortran IV, but the timeless quality of the advice is apparent, and helps to differentiate the substance of style from the fluff of fashion.

The Elements of Programming Style (second edition – I've never come across a copy of the first edition) is a slim and unassuming volume, weighing in at around 170 pages with the title dominating the otherwise plain white cover. This brevity and unpretentiousness reflects the book after which it was patterned, *The Elements of Style* by Strunk and White. *The Elements of Style* offers readable advice for authors (a book worth getting but, as a matter of preference, I would recommend the third edition over the fourth edition). I first read *The Elements of Programming Style* when I was working with FORTRAN. I came across it again a few years later, borrowing it from a colleague. A few years ago, when I started working for myself, I realised that although I quoted and referred to it extensively, I no longer had access to a copy. Some Internet trawling (and surrendering of credit card details) soon sorted that out.

I consider both books chosen so far to be classics. In common with many things that are branded classic, these books are, in computing terms, old. I've also read both of them. And although the Booch book is just under the 600-page mark, the Kernighan and Plauger is a short (re)read. If I'm stuck on a desert island it would be nice to have something to read that was mostly new to me and took a little longer to read. This is the motivation for my third book choice: *Beautiful Code*, edited by Andy Oram and Greg Wilson, contains over 30 contributions from a number of authors (including, yes, a certain Brian Kernighan). I have dipped into it and what I've read so far convinces me that this is one to take away and that it is destined to become a classic. I've found a diverse potpourri of perspectives and examples, good quality writing, some degree of convergence of opinion on what constitutes elegant and simple design, as well as the comfort of reinforcing many of my own opinions – such confirmation always tends to endear a book to the reader!

Beautiful Code draws on a broad range of authors, which helps to guarantee that you get a good mix of examples, large and small, from Rob Pike's svelte `grep` to Google's MapReduce architecture, from the world of testing to the world of language design, from algorithms to operating systems. All too often we talk about learning from our mistakes (which we rarely do as well as we should, which is itself a mistake), but all too often we fail to learn from successes. This is a book of successful designs presented in a way that will leave the reader's mind enlightened and broadened.

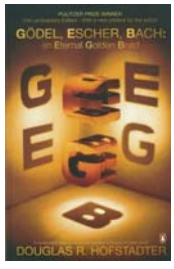
Speaking of incompletely-read anthologies, *ACM Turing Award Lectures* was a close runner up for the island. It contains the papers presented by Turing Award winners from 1966 to 1985 on receipt of their award. It lost out to *Beautiful Code* because, as its more prosaic title suggests, its tone tends to be much drier. However, this book is still worth a mention because it includes insights from people like Tony Hoare, Dennis Ritchie, Ken Thompson, John Backus, Edsger Dijkstra, John McCarthy, Donald Knuth and many more. And for anyone who feels the uncontrollable urge to use the term 'multi-paradigm programming', Robert Floyd's 'Paradigms of Programming' is a must-read. To say of a general-purpose language that it is multi-paradigm is an unsurprising and trivial observation that has little value in distinguishing it from other general-purpose programming languages. This paper provides a useful antidote and course correction to the narrow and casual use of 'multi-paradigm' by being clear about what it means by 'paradigm' (something most users of the word fail to do), and outlines the value of a diverse vocabulary of patterns and styles.

But I digress. If I had been nominating individual papers and articles to take to the island, this would be a different (and even longer) article!

OK, so two books left to choose, one of which needs to be a novel. Rather than choose another book about code and development practice before hitting the fiction, I'm choosing a book that I found profoundly thought provoking and inspiring. It's still geeky, but it's far from being a regular development book or straight computer science tome: *Gödel, Escher, Bach: An Eternal Golden Braid* by Douglas Hofstadter.

The additional caption on the cover hints at the breadth of the book, but perhaps not at its depth or its other qualities: 'a metaphorical fugue on minds and machines in the spirit of Lewis Carroll'. I was first shown and recommended the book by a girlfriend's father – a significant improvement over the traditional shotgun monologue – and eventually got my own copy when I was at university. It is a book of many parts, many perspectives and many narratives, which is probably why for the longest time I used to just dip into it and explore it out of sequence. Eventually, following university, I got around to reading it properly from cover to cover. The way that cognition, computation, philosophy, art, music, mathematics, miscellanea and much (much) more besides are presented is both engaging and enlightening.

Reading the patterns issue of IEEE Software last year, Grady Booch used what I realised was the perfect quote from *Gödel, Escher, Bach* to accompany the discussion of pattern compounds we had in POSA5: 'Fugues have that interesting property, that each of their voices is a piece of music in itself; and thus a fugue might be thought of as a collection of several distinct pieces of music, all based on one single theme, and all played simultaneously. And it is up to the listener (or his subconscious) to decide whether it should be perceived as a unit, or as a collection of independent parts, all of which harmonize.' The only problem being that the ship had already sailed: it was the July/August issue of *IEEE Software* and POSA5 was published in April. Anyway, that's the thing about *Gödel, Escher, Bach*: there is always a new perspective to be found.



In terms of non-fiction, *Gödel, Escher, Bach* had some stiff competition for the island: *How Buildings Learn* by

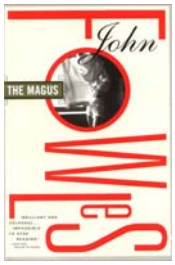
Stewart Brand, *The Timeless Way of Building* by Christopher Alexander, *To Engineer is Human* by Henry Petroski, *The Design of Everyday Things* by Donald Norman, *The Diving Bell and the Butterfly* by Jean-Dominique Bauby and *A Moveable Feast* by Ernest Hemingway. The last two are a little different to the others: they are non-fiction, but they are also non-technical. They would have been candidates for my final book choice, except that no matter how I twisted words and looked for loopholes, all my dictionaries are agreed that a novel must be fiction!

In some ways the choice of novel is perhaps the hardest. There's a lot to chose from, and I've had different interests over time. My habits, preferences and, sadly, time for reading fiction have changed a great deal. The range of good stuff is also a lot broader. To be frank, a lot of technically focused books see and raise Sturgeon's Law (in its abbreviated form: 90% of everything is crap). The functional goal of such books often inspires triteness and poor writing, which makes the good books stand out all the more. Fiction is a more subjective but also less tolerant field.

So, is it to be Hemingway? *The Old Man and the Sea* and *A Farewell to Arms* are great books, but there is something about the ease and situation of *A Moveable Feast* that I particularly like, so if *A Moveable Feast* isn't going to make it to the island, these others won't either. In terms of great American novels, Jack Kerouac's beat classic *On the Road* and the more understated *Dharma Bums* are favourites that fit the escapist feel of the island nicely, as do *The Rum Diary* by Hunter S Thompson, *Zen and the Art of Motorcycle Maintenance* by Robert Pirsig and *The Dice Man* by Luke Rhinehart.

I am also tempted by John Fowles' *The Magus*, a book that has sufficient intrigue and substance to while away the tropical nights. I first came across John Fowles when someone gave me *A Maggot*, a book that almost deserves to be taken to the island solely on the grounds that I think there is more there than I appreciated when I first read it. *The Collector* is a memorable and compelling classic, but such a depressing one that it's definitely staying behind. As I only read it in translation, I am not sure how I would feel about choosing Milan Kundera's *The Unbearable Lightness of Being*. Perhaps the issue of translation is irrelevant, because it was the whole approach that I found refreshing and inspiring. Although good, don't be distracted by the film if you are thinking about this book – quite a different proposition that, by necessity, leaves out a lot of what makes the book the book it is. More recently, Yann Martel's *Life of Pi* is a contender, especially given the castaway situation!

I used to read a lot of science fiction. While there's a lot that I can't see myself revisiting, there are a number of books that stand out. I particularly enjoy Iain M Banks's 'Culture' novels. They are well crafted from the sentence level up to the whole book. *The Player of Games* and *Use of Weapons* definitely deserve a reread. The 'Mars' trilogy by Kim Stanley Robinson – and in particular *Red Mars* – paints a rich picture of something that I used to dream about. A lot of that dreaming was inspired by the writing of the late Arthur C Clarke. Although it had a great effect on me as a child, I don't think that *Islands in the Sky* is something I could return to: more likely *Childhood's End*, *The City and the Stars* or *A Fall of Moondust*. Other notables include *Stand on Zanzibar* by John Brunner, *Ender's Game* and *Speaker for the Dead* by Orson Scott Card and *The Dispossessed* by Ursula Le Guin (this last one recommended to me, as it happens, by the girlfriend whose father recommended *Gödel, Escher, Bach*).



So what's it to be? As I said, this was possibly the hardest choice. On balance, I think John Fowles' *The Magus* – but a last minute repacking is quite possible!

And so to music. I can take one or two albums, which is, as far as I'm concerned, just another way of saying that I can take two albums. It's a desert island, which means sunshine. For me this evokes cold beer, coffee (more than usual), a plausible excuse to wear dark glasses, hay fever and the sounds of Jimi Hendrix, Santana, Led Zeppelin, Placebo, Soundgarden and Jane's Addiction, among others. Something to wake up the island and scare a few animals, basically. Jane's Addiction's *Nothing's Shocking* is a stunning album that still reminds me of when I first saw them live (mumble) years ago. They were just a support act, but I was blown away. However, Placebo's eponymous first album and *Without You I'm Nothing*, their second album, are the frontrunners by the breadth of a vinyl groove. It's a difficult choice because overall I think Placebo's first album is slightly better, but *Pure Morning*, the first track on the second album, is a masterpiece that just tips the balance in favour of *Without You I'm Nothing*.

But there's more to music than disturbing the neighbours. Something else is needed that has a bit of space. Something to complement rather than overwrite the mood of the island; at sunrise, in the lazy afternoon and at sunset. The whole situation puts me in mind of Talk Talk's *Spirit of Eden*, a wonderful and off-beat album that is as far from their earlier chart-friendly sound as my desert island is from civilisation. However, the album is also perhaps too eventful to fulfil the particular role I had in mind. Something with more ambience is needed.

The gradual fade-up of Pink Floyd's *Wish Your Were Here* gently introduces what is perhaps their finest album. There is no denying that The *Dark Side of the Moon* is a great album, but the overall effect of *Wish You Were Here* puts it ahead and on the short list. Dead Can Dance's *Serpent's Egg* is a work of brilliance and an album of the evening. It is easy to imagine the splendour of the opening track, 'Host of Seraphim', as a postlude to the sunset. That soundtrack feel puts me in mind of Philip Glass's *Koyaanisqatsi*. I'm not as keen on his vocal compositions, but his primarily instrumental work is wonderfully immersive. In a similar vein, Steve Reich's *Music for 18 Musicians* is enough of a favourite that I have two different recordings of it. It's evolving rhythm is something I can both lose myself in and work to. Something else I find easy to work, read or zone out to is Brian Eno's *Thursday Afternoon*. It is so gentle that almost nothing happens. Almost. It's brilliant. It's also my second album choice.

Right, I'm off down to the beach to see if I can score some Nike trainers, a satellite dish and a bottle of something decent to drink! ■

Next issue: Allan Kelly (not the Scottish one) picks his desert island books.

Code Critique Competition 52

Set and collated by Roger Orr.

Please note that participation in this competition is open to all members, whether novice or expert. A book prize is awarded for the best entry.

Readers are also encouraged to comment on published entries, and to supply their own possible code samples for the competition (in any common programming language) to scc@accu.org.

Last issue's code

I've written a simple program to count words and it works fine, but when my friend tries it she says it won't compile. Her compiler complains that count is ambiguous [at (1)] and no matching **operator++** found [at (2)]. What's wrong with the compiler?

Can you help answer the question? The code is shown in Listing 1.

Listing 1

```
#include <iterator>
#include <set>
#include <string>
#include <iostream>

using namespace std;

class count
{
  int i;
public:
  count() : i() {}
  void operator++() { ++i; }
  operator int() const { return i; }
};

class word : public string, public count
{
};

ostream& operator<<(ostream& os,
  word const & w)
{
  return os<<string(w)<<": "<<count(w); // (1)
}

int main()
{
  set<word> words;

  word curr;
  while (cin>>curr)
  {
    ++(*words.insert(curr).first); // (2)
  }
  copy(words.begin(),words.end(),
    ostream_iterator<word>(cout,"\n"));
}
```

Critique

Seweryn Habdank-Wojewódzki <shw@agh.edu.pl>

The first thing in all such situations (like this program) is I am using g++ with additional options: **"-Wall -W -ansi -pedantic"**. That, many times, helps.

There are two problems in the code. The first one is that programmer put **using namespace std;** in global scope. This is evil! That's why count class is ambiguous – there exists **std::count** algorithm.

The second problem is that **std::set::value_type** is constant. This is written in Jossutis: *The C++ Standard Library: A Tutorial and Reference* in section 6.10.1 – **container::value_type**. The most important point is that only for **std::set** is **value_type** constant.

So there is no way to execute **operator++** on that value. This is also highlighted by the compiler. The following line simulates what exactly is done in this stage:

```
Words::value_type const & w = (*it);
```

So of course **++(*it)** cannot be done.

There are several possible solutions, here is one that works correctly:

```
#include <iterator>
#include <map>
#include <string>
#include <iostream>

std::ostream & operator<< (std::ostream& os,
   std::pair<std::string, size_t> const & w)
{
  return os << w.first << ": " << w.second;
}

int main()
{
  typedef std::map<std::string,size_t> Words;
  Words words;
  std::string curr;
  while (std::cin >> curr)
  {
    Words::iterator it = words.find (curr);
    if (it == words.end())
    {
      words.insert(std::make_pair(curr,1));
    }
    else
    {
      ++((*it).second);
    }
  }
  Words::const_iterator const end
    = words.end();
  for (Words::const_iterator i
        = words.begin(); i != end; ++i)
  {
    std::cout << *i << '\n';
  }
}
```


ROGER ORR

Roger has been programming for over 20 years, most recently in C++ and Java for various investment banks in Canary Wharf and the City. He joined ACCU in 1999 and the BSI C++ panel in 2002.

He may be contacted at rogero@howzatt.demon.co.uk

Peter Hammond <Peter.Hammond@baesystems.com>

Compiling the original code with GCC (3.4.4 on Cygwin) gives me the following errors:

```
scc51.cpp: In function `std::ostream&
operator<<(std::ostream&, const word&)':
scc51.cpp:22: error: no matching function for
call to `count(const word&)'

scc51.cpp: In function `int main()':
scc51.cpp:32: error: passing `const word' as
`this' argument of `void count::operator++()'
discards qualifiers
```

The first is simply a name clash; the **std** namespace has been 'used' and so **std::count** (from algorithm) is visible, causing the conflict. It is simply avoided by either removing the **using namespace std** or by explicitly scoping **::count (w)**.

The second is more interesting. The compiler is telling you that you are trying to call the non-const method **operator++** on a const object. This is because the iterator returned by **std::set::insert** does not permit modification of the key. A set, like all associative containers, must manage the ordering of its content, which would not be possible if the values could be changed externally though non-cont references. As it happens, you don't actually want to modify the key, but it is a side effect of treating word as both a string (the key) and a count (the value). Now, public inheritance should only be used to model an 'is-a' relationship in an object oriented system. Say to yourself 'a word is a count'. Does that make sense? Not to me it doesn't. What you are trying to do is to *associate* a count with each word, which immediately suggests the use of a map. The following version uses a map to manage counts for each word:

```
#include <iterator>
#include <map>
#include <string>
#include <iostream>

typedef std::map<std::string, int> WordCount;

std::ostream& operator<<(std::ostream& os,
 WordCount::value_type const & w)
{
  return os<< w.first <<": "<< w.second;
}

int main()
{
  WordCount words;

  std::string curr;
  while (std::cin >> curr)
  {
    words[curr]++;
  }

  std::copy(words.begin(),words.end(),
  std::ostream_iterator<WordCount::value_type>
    (std::cout,"\n"));
}
```

The **typedef** just makes it slightly easier to refer to the map's types. The C++ standard [1] requires that a new value created by operator [] is default-initialised (23.3.1.2), which is zero-initialised for an integer (8.1-5), so it can be safely incremented as shown. This all looks nice and neat, with reasonable OO design. The only problem is, it does not compile. It churns out a huge list of template errors pertaining to the fact that **ostream_iterator<WordCount::value_type>** cannot find the **operator<<** that it needs.

At this point in any real-world job I would probably abandon using **ostream_iterator** and use an old-fashioned **for** loop, and I am not alone [2]. In this case the resulting code is considerably shorter, and also the **WordCount typedef** is localised to a smaller scope:

```
#include <map>
#include <string>
#include <iostream>

int main()
{
  typedef std::map<std::string, int>
    WordCount;
  WordCount words;

  std::string curr;
  while (std::cin >> curr)
  {
    words[curr]++;
  }

  for (WordCount::iterator i = words.begin();
    i != words.end(); ++i)
  {
    std::cout << i->first << ": " << i->second
      << "\n";
  }
}
```

Often, however, it is desirable to provide the **operator<<** in a reusable place, and using the more modern, declarative idiom with the copy algorithm can make the code clearer in some cases. So, if you really feel the need to do it that way, you have to put the operator into the standard namespace, where **ostream_iterator** can find it through argument-dependent lookup:

```
namespace std
{
  std::ostream& operator<<(std::ostream& os,
   WordCount::value_type const & w)
  {
    return os<< w.first <<": "<< w.second;
  }
}
```

With this modification, the example above works correctly.

References

[1] *The C++ Standard* (BS ISO/IEC14882:2003), Wiley, 2003.

[2] Verity Stob, 'Out of the (C++) Loop', *The Register*, 2006. http://www.regdeveloper.co.uk/2006/08/08/cplusplus_loops/

David Pol <david@metadev.info>

We will first analyze the roots of the specific compilation problems and look at possible ways to solve them. After that, we will examine the overall design of the program and discuss an alternative implementation based on the use of **std::map**.

I compiled the given program with MVC++ 9, g++ 4.1.3 and Comeau Online 4.3.9. While MVC++ compiled it with no errors, both g++ and Comeau Online gave the same two errors our friend got in her compiler.

Which compiler is right? Well, the answer is that all three are right. The problem lies in the code itself, which was not written with portability in mind.

The first error is easily detectable if you are familiar with the standard library function **std::count()**, and is a clear example of the problems you are potentially being exposed to when writing **using namespace std;** – even in a **.cpp** file – as a lot of identifiers are brought into the global scope (according to 17.4.4.1/2, 'a C++ header may include other

C++ headers', so the contents of the standard library that are effectively made available to the global scope at some point by including a standard library header and writing `using namespace std;` depend on the implementation – in our case, for example, g++ standard library's header <string> happens to include <algorithm>, where the definition of `std::count()` resides). Using-directives help reduce typing, but can potentially create confusion for maintainers and compile-time conflicts due to name clashes. As we have just experienced!

So what can we do? We have several options:

- Change the name of the count type. Or maybe not define this type at all (more on this later).
- Explicitly tell the compiler we want to use the count function that belongs to the global namespace (not the most elegant thing to do, from my point of view):

```
// ...
return os << string(w) << ": " << ::count(w);
// ...
```

- Prefer using-declarations to using-directives, or directly avoid using-directives and explicity qualify identifiers with `std::`.

Regarding the second error, it is necessary to say that the standard delegates the types of `std::set::iterator` and `std::set::const_iterator` to the implementation (23.3.3/2). While some implementations may provide both const and non-const iterators for `std::set`, others do only provide const iterators. So this code

```
++(*words.insert(curr).first);
```

fails to compile on those compilers with a standard library implementation that only provides const iterators for `std::set`. But we may really need to modify an element in a `std::set` in a portable way (more precisely, a non-key part of an element in a `std::set`, as altering a key part of the element could break the sortedness of the container).

So, what can we do now? Again, we have several options:

- Use `const_cast` (but remember that casts are dangerous and it is better to avoid them whenever possible):

```
++(const_cast<word&>
(*(words.insert(curr).first)));
```

- Make the member variable `i` inside count mutable and define count's `operator++` as a const member function (this illustrates the fact that code that works does not always make sense at all; in our case, I consider this to be a non-solution and a clear indicator that `std::set` is very probably not the way to go).
- Use the safe way of changing elements in a `std::set` described by Meyers in *Effective STL*. It consists on locating the element to modify, make a copy of it, erase the element from the container, modify the copy and insert it into the container [*]:

```
// ...
while (std::cin>>curr)
{
  std::set<word>::const_iterator it =
     words.find(curr);
  if (it != words.end())
  {
    word temp(*it);
    words.erase(it++);
    ++temp;
    words.insert(it, temp);
  }
  else
  {
    words.insert(curr);
  }
}
// ...
```

  [*] Note that we would also have to initialize the member variable `i` inside `count` to 1 in this case.

- Separate the element type into a const part that determines the ordering (the key) and a mutable part that does not determine the ordering (the value), and use a `std::map` that maps keys to values.

Now that we have discussed ways to make the code portable across compilers, we proceed to question its current design. We want to count words, so we clearly need an association between a given word and its number of occurrences. Does not that make you think almost immediately about `std::map`? We are not really getting that much from using `std::set`, as we need to create two additional classes, one of them publicly inheriting from `std::string` for convenience (although standard library container classes are not intended to be publicly derived from and it is usually preferred to use composition or private inheritance instead).

Using `std::map`, our program looks like this:

```
#include <iostream>
#include <map>
#include <string>

int main()
{
  typedef std::pair<std::string, unsigned int>
    word_occ_pair;
  typedef std::map<std::string, unsigned int>
    word_map;
  word_map words;

  std::string current_word;
  word_map::iterator it;
  while (std::cin >> current_word)
  {
    it = words.find(current_word);
    if (it != words.end())
    {
      words[current_word] = ++(it->second);
    }
    else
    {
      words.insert(word_occ_pair(
        current_word, 1));
    }
  }

  word_map::const_iterator end_it =
    words.end();
  for (word_map::const_iterator it =
       words.begin();
    it != end_it; ++it)
  {
    std::cout << it->first << ": "
      << it->second << std::endl;
  }

  return 0;
}
```

We can see that we are able to avoid major sources of problems by using the right container from the beginning.

Ivan Uemlianin <ivan@llaisdy.com>

Overview of the problem – two sets of problems

The purpose of the submitted program is to take some text from standard input and output a list of word counts. It compiles under MS Visual C++ and works correctly, but g++ will not compile it. We are asked, 'What is wrong with the compiler?' Obviously it is the compiler which is at fault.

There are two sets of problems with this code. The first is to do with compiler sensitivity: the program will compile under at least one compiler, but not with at least one other. The second, more fundamental, set of problems, of which the first is probably a symptom, is to do with choice of central data structure.

Problem 1: compiler sensitivity

The program will compile and operate correctly under MS Visual C++, but g++ will not compile it, reporting errors on two lines. One possible response to this would be to say, 'Well, VC++ is obviously a better compiler, and I only want to run on Windows anyway.' Or perhaps we could run a survey of C++ compilers and see 'empirically' how the majority of compilers treat this code.

C++ compilers do behave differently, and comparison is often worthwhile. However, most code should compile perfectly well on all compilers. If code this simple is showing compiler sensitivity, then the problem is more likely to do with the code than with the compiler. So, where does g++ have a problem?

count(w)

The first error is on line 24 in the `operator<<()` overloading:

```
ostream& operator<<(ostream& os,
                    word const & w) {
return os<<string(w)<<": "<<count(w); \\ (1)
}
```

g++ says:

- on cygwin:

```
no matching function for call to 'count(const
word&)'
```

- on linux:

```
reference to count is ambiguous
candidates are: class count
/usr/include/c++/4.2/bits/stl_algo.h:424:
error: template<class _InputIterator, class
_Tp> typename
std::iterator_traits::difference_type
std::count(_InputIterator, _InputIterator,
const _Tp&)
```

`count(w)` seems to be a simple typo for `int(w)`, as `count::int()` is not used anywhere else in the program, and the point of `count(w)` in line 24 is to return the `int i`. The line will compile under g++ with `int(w)` in the place of `count(w)`.

How come it compiles under VC++? This is mysterious to me. I imagine VC++ is treating `count(w)` as a cast (and `(count)w` as the same effect here). The word `w` is cast to a count, the only printable part of which is the `int`. However, I haven't been able to find any worthwhile documentation on casts in VC++.

words.insert

The second error raised by g++ is on line 39 in `main()`:

```
while (cin>>curr)
{
  ++(*words.insert(curr).first); // (2)
}
```

This is equivalent to:

```
while (cin>>curr)
{
  pair<set<word>::iterator, bool> p =
     words.insert(curr);
  ++(*p.first);
}
```

And, on g++, it is the second line of the loop which fails, with:

```
passing 'const word' as 'this' argument of
'void count::operator++()' discards qualifiers
```

If we replace `(*p.first)` with `(p->first)`, we get a more informative error from g++:

```
base operand of '->' has non-pointer type
'std::pair<std::_Rb_tree_const_iterator<word>,
bool>'
```

So under g++ the first element of the pair is actually a const iterator – not so under VC++ (although of course `p->first` still raises an error).

I don't know how to fix this, but in any case fixing compiler sensitivity is not the way to fix the program.

Problem 2: the wrong data structure

The main problem with this program is that it uses the wrong data structure. The purpose of the program is to collect frequencies of words in a text, in other words to provide a mapping between a word and its frequency. For this task the correct data structure is a map, not a set. Much of the complexity of the program is caused by the programmer trying to cope with this fundamental error.

Correcting the data structure

Using `<map>` instead of `<set>` the program becomes much simpler:

```
#include <iterator>
#include <map>
#include <string>
#include <iostream>

using namespace std;

void output(map<string, int>& words)
 // cribbed from tc++pl p481
{
  typedef map<string, int>::const_iterator ci;
  for (ci p = words.begin(); p != words.end();
      ++p)
    cout << p->first << ": " << p->second
      << \n";
}

int main()
{
  map<string, int> words;
  string curr;
  while (cin>>curr)
    ++words[curr];
  output(words);
}
```

No classes, no operator overloading, and the output function is cribbed (from a reputable source [1]). The program is clear and concise, and VC++ and g++ both compile it without complaint.

Conclusion

This program started going wrong when the choice was made to base the solution on sets. From that false step, complications pile on complications. Look after the data structures, and the algorithms will look after themselves.

References

[1] Stroustrup, B. (2000) *The C++ Programming Language* Addison-Wesley, ISBN: 0201700735.

Nevin :-] Liber <nevin@eviloverlord.com>

What's wrong with the compiler?

I don't know if there is or is not anything wrong with the compiler. In this case, the code, not the compiler, is the problem.

Issue (1): `using namespace std;`

This has a very powerful effect: every symbol in `std` that is used in this translation unit (file) can be used without having to qualify it with `std::`.

In <algorithm>, there is a function named **std::count**. Now, while the author isn't directly including it, some (but not all) implementations of **std** library components do include it, hence causing the ambiguity.

Issue (2): In order to maintain sorted order, the members of **set<Key>** are of type **const Key**, and non-const member functions (such as **operator++**) cannot be called on it.

A better solution:

```
#include    <map>
#include    <string>
#include    <iostream>

int main()
{
  typedef std::map<std::string, int>  Words;
  Words                               words;
  Words::key_type                     curr;
  while (std::cin >> curr)
    ++words[curr];
  for (Words::const_iterator i =
       words.begin(); i != words.end(); ++i)
    std::cout << i->first << ": " << i->second
      << '\n';
}
```

A few notes:

1. I used a **map**, not a **set**, to allow modification of the count. The negative to this is writing an explicit for loop for output.
2. Instead of a **using** statement, I fully qualify all my uses of things from **namespace std**. Another alternative would be a **using** statement for each of the types one wished to use, as in:

```
using std::map;
using std::string;
using std::cin;
using std::cout;
```

Commentary

I'm glad to see that reverting to the more usual C/C++ code critique seems to have attracted more interest.

This problem is interesting because there are two different compiler-specific problems with the code. It can be very hard to detect in advance places where you've relied on compiler-specific behaviour and it can be quite hard to resolve the resultant problems. This is something where greater experience helps, and also using a high compiler warning level.

As a rule, if you are writing code that needs to be portable, the best technique is to compile from the outset with a wide range of compilers. This finds problems up front, but also educates you about potential trouble spots.

The Winner of CC 51

It was hard to pick a winner this time – partly because there was a lot of overlap between the entries. Thank you to you all for contributing to this column! I particularly liked Ivan's use of the output helper function which I thought helped make his solution very clear.

I have awarded this issue's prize to David Pol – in my view his critique best provided both clear explanations of what was wrong and also listed a good variety of possible solutions.

Code Critique 52

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

I am having problems getting a template to work. The code below is supposed to print out the range of the data points, but the range function isn't producing the right answer for the WDatum class. Someone told me it was because I needed a virtual destructor – but that made it worse. I've even got rid of all the compiler warnings I had. Can you help me understand what's gone wrong?

Listing 2

```
// Simple datum
class Datum {
  float payload;
public:
  Datum( float value = 0 )
  : payload( value ) {}
  float getValue() const
  { return payload; }
};

// Weighted datum, simple by default
class WDatum : public Datum {
  float weight;
public:
  WDatum( float value = 0, float weight = 1 )
  : Datum( value ), weight( weight ) {}
  float getWeight() const
  { return weight; }
};

// Return range (max - min) of data
template <typename T>
float range( T * begin, T * end )
{
  float top = 0, bottom = 0;
  for ( Datum *it = begin; it != end; ++it )
  {
    float v = it->getValue();
    if ( !top && !bottom )
      top = bottom = v;
    else if ( top < v )
      top = v;
    else if ( bottom > v )
      bottom = v;
  }
  return top - bottom;
}

#include <iostream>

int main()
{
  static const int count = 4;
  Datum data[count] =
    { 1.3f, 1.2f, 1.4f, 1.7f };
  WDatum wdata[count] =
    { 1.3f, 1.2f, 1.4f, 1.7f };

  float drange = range( data, data+count );
  float wrange = range( wdata, wdata+count );

  std::cout << "range (expect 0.5)\n"
    << "Datum " << drange << "\n"
    << "WDatum " << wrange << std::endl;
}
```

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

Regional Meetings

A round-up of the latest ACCU regional events.

ACCU London

Report from Steve Love (steve@essennell.co.uk)

The ACCU London chapter is now well established and holds regular monthly meetings. Actually, our aim is to hold nine meetings a year with a featured speaker, plus a Christmas social event but no meetings in August (because it's summer and everyone wants a holiday) and to skip the March or April meeting depending on which is closest to the annual conference.

In the past year subjects have included: C++ 200x, what's new in Java, introducing Agile development, product management and the world of recruitment consultants.

Meetings are usually held at the offices of 7 City training (http://www.7city.co.uk) on Chiswell Street in central London (i.e. the City). We are very grateful to 7 City for providing these excellent facilities for free. As so many of our members (and potential members) work (or even live) in the Docklands area, we aim to hold occasional meetings there. On these occasions we rely on the hospitality of other institutions: so far both Barclays Capital and Lehman Brothers have provided rooms and we are grateful to both.

Meetings are normally held on the third Thursday of the month. Occasionally we have to move the meeting to another date, but still we aim to keep them on Thursdays. All meetings are open to members and non-members alike – although we hope non-members will join – and are free of charge.

Upcoming meetings are:

- 19 June: Microsoft, New Stuff in C# and Linq
- 17 July: Jason McGuiness and Colin Egan, The Challenges facing Libraries and Imperative Languages from Massively Parallel Architectures
- August - no meeting
- 18 September: Andrew Holmes, Introduction to Value at Risk
- 16 October: TBA
- 20 November: TBA
- 12 or 19 December: Social event, TBC

To receive notices about upcoming ACCU meetings in London, subscribe to the accu-london mailing list by sending an empty email with the subject 'subscribe' to accu-london-request@accu.org or by visiting http://lists.accu.org/mailman/listinfo/accu-london. The list is open to non-members and is very low volume.

More information and an up-to-date schedule are available on the ACCU website: http://accu.org/index.php/accu_branches/accu_london

If you have any questions or suggestions – particularly for future speaker or location – please contact either James Slaughter (slaughter@acm.org) or Allan Kelly (allan@allankelly.net).

ACCU Cambridge

Report by Pete Goodliffe (pete@cthree.org)

Thursday 1st May 2008

It's been a while since the last ACCU Cambridge meeting, and this one was a definite return to form!

ACCU stalwart, Roger Orr (from sunny Londonshire) took us on a tour of the joys, intricacies, and new toys coming with the forthcoming C++ "0x" language standard. As you'd expect from Roger, the talk was interesting and entertaining. The venue (again, the DisplayLink offices situated on the nearest thing to a hill we have in Cambridge) was packed, almost exclusively with C++ programmers who clearly wanted to know the new ways they will be able to write fascinating bugs once compiler vendors catch up with the ISO committee (because we all know what a breakneck speed the ISO standards are produced at).

The evening concluded with traditional après-talk beers at the Castle.

It was another excellent evening – many thanks to Roger for speaking and Ric for organising.

Bookcase

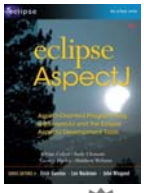
The latest roundup of book reviews.

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

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

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

Eclipse and AspectJ



by Adrian Colyer, Andy Clement, George Harley and Matthew Webster; published by Pearson Education, Addison Wesley; ISBN: 0-32-124587-3

Reviewed by Omar Bashir

Highly recommended.

One of the simplest ways of judging a book on advanced concepts in computer programming (like aspect oriented programming) is to see how early and how comfortably one can start programming while reading that book. This book certainly allows readers to start experimenting with aspect oriented programming very early and ensures that the entire experience of getting accustomed to aspect oriented programming is painless and exciting.

The readers are expected to be familiar with object oriented programming in Java and the use of Eclipse as a Java IDE. It is an extremely well structured book, divided into three main parts. The first part introduces AspectJ as a language to incorporate aspect orientation in Java. The book proceeds with an explanation and rationale of aspect orientation, installation of Eclipse and the installation of AJDT (AspectJ Development Tools) plugin for Eclipse. The second part of the book provides a comprehensive AspectJ language reference. Finally, the third part of the book focuses on advanced topics, AspectJ adoption strategies and aspect oriented design.

The book explains various concepts of AspectJ using an example that is progressively refined as the book progresses. The examples based on a simple insurance applicatio, are mostly easily understood. In addition to the source code, authors also provide various UML diagrams describing the structure and dynamics of the application. Authors also describe and illustrate the various facilities provided by Eclipse to support AspectJ development, which include various icons that appear on the classes, methods, aspects, pointcuts and advices in the Editor, Package Explorer and Outline views. Additionally, a new Visualiser view is introduced that provides a display of all the classes in the project and the approximate locations of advices and inter-type declarations.

One of the most interesting features of this book is a fairly detailed recommendation of a process of adoption of AspectJ in Java projects. The process starts with individual adoption of AOP and proceeds through aspect libraries shared and reused within and across projects. At the individual level, the aspects used are mostly enforcement aspects that can be used to implement rules within the projects. Moving forward, infrastructure aspects can be used to provide general cross cutting facilities within the project. With more experience, core business aspects can be added to the project and the final phase of adoption of AOP is based on developing aspect oriented architecture and design of the systems being developed.

Advanced concepts in the book also touch upon building aspect libraries, linking to compiled class files and integration with Ant. Although the book is not an aspect oriented design book, the authors do provide a brief introduction to aspect oriented design. This includes ways to identify aspects within an application by analyzing requirements.

A few examples in the book are based on Hibernate and Spring. Although these technologies are well known but it is not necessary that most developers interested in AspectJ are experienced in these technologies. Thus, understanding these examples may require understanding these technologies, which may not be possible in cases.

This book is strongly recommended as a reference to AspectJ and also as a comprehensive tutorial for beginners in AspectJ.

Scaling Software Agility: Best Practices for Large Enterprises

by Dean Leffingwell, published by Addison Wesley (2007), ISBN 0-321-45819-2

Reviewed by Omar Bashir

This book is by far the most interesting text I have read on software development processes and methods. The overall objective of this book is discussing the applicability of agile methods in large enterprises and projects. The book is based on experiences of the author in applying agile development processes and methods in projects and organizations of varying sizes and complexities.

The book is divided into three parts. The first part describes the established and practiced agile 300 methods. The focus here is on identifying aspects of these methods that enable and promote agility and also towards highlighting common features of these methods. Part two of the book discusses all the agile team practices that scale. Interestingly, with slight adaptations, most agile practices can be applied to medium and large sized projects. Finally, the third part of the book discusses organizational issues in implementing agility at an enterprise level.

Instead of diving straight into agile methods, the author discusses the classical waterfall model and the reasons for it not delivering quality software in time and on budget. The author describes at length the impact of prolonging the release schedule on the variation of user requirements. The author describes in detail how most applications of the waterfall model enhance the risk in the software development process because of isolated development, late integration, fewer (usually only one) release and inadequate testing.

Bookshops

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

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

The author also counters various misconceptions the general community has about agile development. He stresses that agile is not at all ad-hoc. On the contrary, success with agility requires planning and managing the development process. The text highlights that agile is not also uncontrolled but iterations and release cycles consist of definite activities that need to be performed to achieve successful delivery of the project. For successful agility within a development organization, the organization needs to evaluate each iteration and release cycle to determine aspects in the process that can be further improved. Furthermore, the author also underlines the need for system architectures that support agility. The most interesting aspects of the book include managing agility in large distributed teams, topics that are not widely discussed in most other agile development texts. The text tends to lean more towards Scrum and RUP however it does not disregard other methods. The author does discuss the application of various practices from other methods in specific contexts.

The book contains very relevant and descriptive illustrations that are very helpful in explaining the text. The author also explains various tools that assist in applying agile methodologies. The text is well written, focused and descriptive without being verbose. The book includes examples and case studies from author's experience in implementing agility in a number of different development environments and enterprises.

I strongly recommend this book to all involved in software development. Not only does it provide an excellent introduction to agile methods for beginners in the field but it also highlights practices that can be applied to a wide range of development environments. In fact reading this book before reading text on any other agile method may help the reader to understand shortcomings of the method under study.

Provably Correct Systems: Modelling of Communication Languages and Design of Optimized Compilers

by He Jifeng; C. A. R. Hoare; and Jonathan Bowen; published by McGraw-Hill (1995), ISBN 0077090527

Reviewed by Colin Paul Gloster

This is not an unpleasant book to read but it lacks outstanding attributes to compel you to read a copy (unless you can acquire it cheaply and easily). As with maybe any book, it has some flaws (I leave it to you to decide whether you would classify defining a sans serif T on page 86 to be `false` to be one of the book's flaws, and I assure you that that was not a typo), not least of which is that the coauthors did not seem to be clear as to what they were aiming to accomplish. The subtitle may seem to indicate that you will learn about how to write an optimizing compiler, but instead a rapidly developed prototype compiler is presented in the final chapter with the suggestion that the 'prototype compiler could be used as a compiler checker to compare the output generated by a particular compiler implementation'. Nothing essential for writing a compiler (even one which does not optimize) can be learnt from this book. Why a compiler implementer's view is unimportant on page 84 is never explained: 'Of course, in the actual implementation of the compiler there may be many internal interfaces, for example between successive compilation passes and the loader. Such interfaces are of no concern to any user of the compiler, and are not treated in this book.' Whether the coauthors really understood why compilers are popular is doubtful: e.g. though valid reasons are listed as to why to use a high-level language (HLL) instead of machine code (often called 'machine language' in this book), two of the most important were omitted viz. a person writes the same quantity of lines of code per day independently of language whereas a compiler will typically generate more than one machine code instruction per HLL line; and compilers often optimize. Furthermore, the presented unfinished compiler checker is not proven to be correct, despite the title and the promise in the blurb.

In the final chapter, Prolog's backtracking was claimed to be efficient (!); early in the chapter there is some interesting advocacy of logic programming instead of functional programming (which seems to have been forgotten by the end of the chapter when Prolog is compared only with imperative languages); along with some unintentional bad advertisements of Prolog such as Prolog source code must be delicately laid out such that 'variables are properly instantiated before the checking clauses are invoked'.

Section 8.2 Compilation of Expressions contains an unintentional formal error in 'The most commonly used arithmetic operators in programming are `+`, `-`, `*` and `\`' (i.e. \ instead of `/`). The remarkably uncommon opinion 'The richness of a programming language is determined by its operators' was espoused by coauthor He Jifeng in Chapter 8. Operators were not among the highlights chosen for the section 1.1 Why ML? of the book *Elements of ML Programming* even though ML is much more advanced in this regard than mainstream languages and Prolog's support for user-defined operators is even more advanced than ML's and put to good use in Chapter 10 by coauthor Jonathan Bowen, but I disagree that a language is characterized overall by its operators.

Coauthor C. A. R. Hoare's renowned reputation is bolstered by predictions which he made in his chapter which came true but is blemished by mistakes on page 3. He incorrectly claimed `value_after > value_before` 'describes the behaviour of any piece of code which does not decrease the value'. It does not hold for an identity such as `getBankBalance()`. On that page he nobly but naively claimed 'Since all kinds of failure are to be avoided, there is no need to make fine distinctions between them, or to give accurate predictions of the behaviour after failure'. Fault-tolerance is a major issue in aerospace and I had been offered two Ph.D. positions including one scholarship for concentrating on one specific type of error which was discovered due to a fine distinction.

Elsewhere in his chapter, Hoare used two pages to explain why we should resign ourselves to the notion of tolerating non-determinism. The justification is not clear until near the end of the two pages, by which time it is almost convincing.

The index is useless.

SAMS Teach Yourself Django in 24 Hours

by Brad Dayley, published by SAMS (2008), 507 pages, ISBN 067232959X

Reviewed by Ivan Uemlianin

Highly recommended

I was recently forced to be away from my computer for a week or so (not on holiday!), but knew that on my return I should be working on Django, the Python web development framework [1]. I found this book in a local bookshop and bought it despite the variable reputation of SAMS' books.

I'm glad I bought it. I found it very readable and clear and despite 'life' happening around me I got a good grasp of Django theory and practice. Now I've been working on Django a bit I still find this book handy. Even though Django's own documentation is very good indeed, I highly recommend this book as a quick start guide, especially if you're offline or you just prefer paper.

The 24 'hours' of the title are the 24 chapters of the book, each of which is supposed to take an hour to read and work through. This is either a swizz, corny or quite clever depending on your mood – and on how good the book turns out to be.

The first 12 chapters cover material essential to understanding Django, describing the way Django uses 'models' and 'views', and how to implement a basic Django web application. Chapters 13-23 discuss various common but not stricly essential features like user sessions and security, cookies, caching, internationalisation, etc. The final chapter covers deploying Django under a web server like apache. It's quite appropriate to have this at the end, as most development and testing of a Django app. is done under Django's own lightweight web server.

Chapters have 'Try it yourself' sections (more in the early chapters) which guide you by the hand through implementing the procedures described. These are not exercises with questions and answers (although there are 'quizzes' at the end of each chapter) – you are told exactly what to do, and full listings show exactly what you should end up with.

[continued on back page]

ACCU Information

Membership news and committee reports

View From The Chair

Jez Higgins
chair@accu.org

Each year at the AGM, ACCU has the opportunity to confer honorary membership to those who have made a particular and long-standing contribution to the organisation. This year the AGM recognised Ewan Milne, for his work as conference chair. Giovanni Asproni, our current conference chair, delivered a warm appreciation of Ewan's work for ACCU. Ewan chaired the conference committee for four years, from 2004 to 2007, delivering strong programme after strong programme. He was instrumental in bringing many high profile speakers to the conference, some rarely seen outside the US. He also oversaw the introduction of the pre-conference tutorials, which have given many people opportunities to learn from some of the best in our industry at extremely reasonable cost. Ewan was my predecessor as ACCU Chair, a position he held between 2003 and 2006. His honorary membership is richly deserved.

Many thanks to those who were able to attend the AGM. On behalf of the officers and committee I would like to thank you for electing or re-electing us, and for the confidence you show in us. As he notes in his own report, Allan Kelly has taken over from John Merrells as Publications Officer. John held that post for a number of years, and is also a former editor of Overload. Allan suggests the position is one of long periods of nothing punctuated by the occasional crisis. John's good in a crisis and I would like to thank him for his help over the years. I'm sure Allan has an equally safe pair of hands.

In fact, it's change all round for our publications. Alan Griffiths, another former Chair, recently handed the Overload editorial reins to Ric Parkin. In my time on the committee, Alan has quietly and reliably got on with Overload, overseeing solid issue after solid issue. As Chair, I've valued his sensible and helpful contributions to committee discussions. Here in CVu, we're looking for a new editor to take over from Tim. Tim has done a cracking job on CVu, and he is leaving it in a very healthy state.

ACCU is very lucky, I think, to have the benefit of people like Ewan, John, Alan, and Tim. We don't always express our appreciation as often or as vocally as we should, and I include myself in that. As an individual member, I'd would like to publically thank them all for their work. Their help has made me a better programmer. As Chair, I would like to thank them on behalf of the membership for making ACCU a better organisation for all of us.

Membership Report

Mick Brooks
accumembership@accu.org

I enjoyed the chance to speak to many members at this year's conference. I asked some of you about what you value about your ACCU membership, and about your experiences introducing the organization to friends, colleagues and employers. Overwhelmingly, members were enthusiastic about the community aspect of the organization: they feel that by subscribing to ACCU they support, and become a part of, a group that stands for constant learning and improvement. This intangible benefit was mentioned far more often than the tangible magazines and mailing lists, and even the conference discount.

However, this is clearly a more difficult sell to others (particularly employers), something a number of you found, and of which I know I don't make the most when talking to prospective members.

Not everybody can make it to the conference, and there's not enough time between sessions to speak to the majority of those that do. I'm keen to hear all your ideas about how best to promote ACCU and grow the membership, and any of your experiences of trying to do so; please email me (accumembership@accu.org) with your thoughts. I'm also interested in how we can help you to introduce us to the people you know (if you want to give it a go, I usually have spare journals and flyers that can help).

As ever, contact me for any questions about membership, journal addressing and renewals.

Publicity Officer Report

David Carter-Hitchin
publicity@accu.org

It's been a good month or two for publicity. We have been approached by Bernard Opic and Art Mealer to setup new ACCU chapters in their countries (France and USA respectively). If you are reading this and you are outside the UK, then please think about ways to promote the ACCU in your country. I can provide some help and advice, so please mail me. The work that Bernard and Art are doing is great news for the ACCU. Bernard is also going to translate some articles which will help with the language barrier. Hats off to Bernard and Art.

I recently emailed all our student members and a leaflet should be on their college notice boards now. Nearly everyone I emailed responded positively, but I'm acutely aware that this is only a fraction of the overall student population. If you are a student and haven't been in touch with me then please do so and I'll tell you how you can help. If you know any students who are not ACCU members but could put up some leaflets, then let me know.

On the other side of the academic coin, the information about Computer Science/Physics/Engineering/Maths courses and tutors in the UK has been assembled and I'm about to mail out our publicity letter. This is a big undertaking, with the letter being sent to about 250 academics.

In the next year I want to see an ACCU leaflet up in every library in the UK. This sounds like a massive task, but it's relatively easy, actually. All I need is one or two volunteers in each county/region. Please send me an e-mail about this and I'll tell you the next steps. Basically ALL you will need to do is talk to your local library and ask them if they can distribute leaflets to all the other libraries in your county, and if so how many they need. The leaflets will then be sent to you to give to your library. I'll keep track here of which counties have been covered. Easy! Please volunteer for this by sending me a quick email NOW.

Finally, I'd like to mention conferences and the visibility of the ACCU at them. In an ideal world, we would all have tons of time and money to go along to other programming and IT conferences, but sadly we live in a world very much constrained by time and money. This said, if you are going to a conference, please arrange to bring along, at the very least, a bunch of flyers that could be put on an information table. Also, if you are speaking, make sure to mention the ACCU! Please mail me with any conferences you know about at which would be good to find members.

Finally, did I ever mention that you should put http://www.accu.org/ in your e-mail signatures?

Publications Officer

Allan Kelly
publications@accu.org

As those of you who attended the ACCU AGM will know, I was elected to the sleepy backwater post of Publications Officer. I agreed to accept the nomination for this post because I knew, or thought I knew, just how little work this post entailed. Apart from sorting out the occasional publication production glitch or finding a new journal editor every couple of years there is nothing to the job. After all, the hard work is done by the CVu Editor, the Overload Editor and our fantastic Production Editor, Alison. Little did I know…

Alan Griffiths was already in the process of handing Overload over to Ric Parkin before anyone mentioned Publications Officer to me. In a way I'll not be sorry to see Alan go from Overload, he's done sterling work for the last few years as editor but I've missed his articles. I hope he'll now find the time to return to writing.

Secondly, from time to time it is worth changing editors to keep the journals fresh. Alan has outlasted two CVu editors. I am confident that Ric is going to make an excellent editor.

What I didn't know (until too late) was that Tim Penhey also wants to step down as CVu editor. Since taking the job, Tim has relocated back to New Zealand, taken on a demanding (paid) job and, recently, signed a book contract so perhaps it's understandable that his time has come. Personally I'm sorry to see Tim go, under his editorship I think CVu has reached new highs, for me it has never been better.

Tim mentioned to me that he was disappointed not to have had more feedback from readers about CVu. He once hoped to start a letters page but it never got very far. I have sympathy with Tim here. During the years I've been writing for the ACCU (those with long memories may recall that I contributed to almost every issue of Overload between mid-1999 and mid-2004) I have had very little feedback or comments from readers. In fact, I can confidently assert that I don't need all my fingers to count the letters, e-mail and comments I received.

Now I make an effort to send a thank you e-mail to writers when I particularly enjoy their articles – especially when they are first time writers. I hope that more members will try to follow this example. If you enjoy something just send a 'thank you' note.

And if you read something that makes you think, please let the author know and send a letter to the journal editor with your comment. I'm sure we could easily fill a letters page if we tried.

We now need to find a new CVu editor. If you are interested please let me know. For the next few issues we have decided to try something new. Tim has agreed to stay on for a while as the Executive Editor while a number of different people take a go at being guest editor. Jez Higgins, Guy Bolton-King, Gail Ollis and Roger Orr have all offered to guest edit an issue. So again, if you are interested in guest editing an issue please get in touch.

I hope that one of our guest editors will eventually take over a full time editor. Indeed, if we have many volunteers to guest edit we may even appoint a new editor and continue with guest editors.

There is only one rule I will lay down for editorial appointments on my watch: you cannot have held the job before. As I said above, I think a new editor should bring something fresh.

To me one of the great things about the ACCU is that we are constantly asking our members to step up and contribute. Whether by writing for the journals, editing the journal, joining the committee, organising the conference or running a mentored developer project there are always opportunities to contribute. To an outsider it might look like we are short of contributors but I don't see it that way, to me we are long on opportunities.

So if an ACCU hand taps you on the shoulder one day and says 'Have you ever thought of….' just remember that someone has noticed you and thinks you have what it takes to make a contribution. In making that contribution you will grown; you will learn and you will improve.

I digress, there is another change on my plate even bigger than the CVu editorship we need to discuss.

Currently the ACCU publishes twelve journals a year: these come in pairs every six months. More and more people have been wondering why we don't publish one a month. At conference I spoke to a lot of people concerned with journal production and everyone felt the time was ripe to change. So we are going to change.

The current plan is to delay the October CVu by one month, make it a November CVu. From then on, Overload will continue to appear in February, April, June, August, October and December, while CVu appears on the alternate months. This also means that March CVu can become a 'Pre-conference special' issue.

This will cost the ACCU more money, both production costs and postage costs, but I am not alone in thinking this is worthwhile. First of all, there will be more advertising slots to fill so we should find more revenue. Second we will give our members a regular reminder that we are here and help grow our membership. Finally I think it will help the journals assert their own identity further.

Some people have asked why we have two journals, why not have one? Well we might decide to do that in future but right now the simple answer is: we have two journals because we have two journals. It might be that in time the journals converge and we decide to merge them. That would create more work for one editor; with two journals we split the work between two people.

I hope this will be my first and last report as Publications Officer and in a month or two it can revert to a nice backwater post. With that said please get in touch if you would like to edit or guest edit CVu and please, if you like an article send a thank you e-mail to the author (the journal editor might appreciate one too!) ■

Book Reviews (continued)

I found typo and broken code density to be remarkably low. The prose is simple and direct, without the self-consciouness of 'Dummies'-type books. Using the book was straightforward, and I feel I understand the material.

It could be argued that the full listings and copious screenshots act to pad out what is really a fairly slim book. It didn't feel like that to me. As a reader, I felt mollycoddled – in the best sense of the word. The amount of scaffolding and safety nets meant I could just relax, read and absorb. Too many computer books are badly written, incoherent and just hard work to decode.

This is a very good first book on Django. It will happily fit into whatever space you have in your life, and it will take you from knowing nothing to being able to write an application for yourself, or to set up one of the many documentation-less Django projects on the web.

[1] http://www.Djangoproject.com