The magazine of the ACCU

www.accu.org

{cvu}

Volume 20 Issue 2 April 2008 £3

Features

Regulars

{cvu}

Volume 20 Issue 2
April 2008
ISSN 1354-3164
www.accu.org

ACCU – what's it all about?

I don't know about you lot, but I tend to enthuse about ACCU when I'm around other technical people. Then I get hit by the question "What's it all about then?" This is where I normally go, um... ah... professionalism in programming. What does that really mean though? I feel that we need something a bit more than that in order to explain what ACCU is all about.

So here I am, sitting talking with my wife, trying to work out exactly what ACCU gives you. Well obviously it gives you C Vu, and Overload if you pay that little bit extra. There are mailing lists, but to be honest, there are heaps of places that have mailing lists. ACCU has a conference – a damn fine conference if you ask me. Unfortunately things didn't work out for me this year and I was not able to make it. I've heard that ACCU has a good standing with several technical book publishers as we tend to write good quality book reviews. We are getting more local meetings, and that's a really good thing. And that is close to where I start losing it.

Unfortunately books will only be sent to reviewers in the UK, and if you can't get to the conference it seems like there isn't a huge benefit. I personally find myself very connected to ACCU, but I just can't seem to pass that connectedness on to people that I talk to about it. Why is that? Are they stupid? Don't they get it? I think the answer is "No, they don't get it" but we don't seem to be very good at helping them get it.

After more talking trying to identify what I felt was the crux of ACCU, we came up with this. ACCU is about getting around other people who are technically as good as, or better than, you. Being around people like this makes you realise that you still have a lot to learn and a lot of space to improve. Some people really don't care about improving in their profession, and those people we can't really help. But the people who do want to improve, those who do care, these are the people we need to reach out to. Let them know that there is a whole community of other people like them, and they can be found in the bar^W^W^W at accu.org.

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 20.3: 1st May 2008
C Vu 20.4: 1st July 2008

IN OVERLOAD

Stuart Golodetz continues his 'Watersheds and Waterfalls' series and Richard Harris introduces a new problem to his Model Student series: 'A Knotty Problem'.

ADVERTISE WITH US

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

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

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

COPYRIGHTS AND TRADE MARKS

The Town Planner's Triumph

Pete Goodliffe concludes his mini-series on software design.

You can't have too much of a good thing. If you have too much fun, you get tired. If you have too much alcohol, you fall over. If you smoke, you get cancer. If you travel on gas-guzzling transport, you cause global warming. If you eat too much fine food and sweet puddings, you get fat. If you have too much cheese, you can't sleep. If you're too selfish, you have no meaningful relationships. If you have chemotherapy, you lose your eyebrows. If you tap dance in the centre of a department store, you get funny looks.

Life is hard, isn't it? We're constantly trading off one thing for another.

This holds for design as much as anything else. Design is a process of making tradeoffs between competing forces, and trying to trick the laws of nature into allowing you to craft the sublime out of the complex. It's hard work.

This explains why so much bad design exists. It explains why I've never yet seen a teapot that doesn't drip. And it explains why some people sitting in a stadium will always have their view obscured by a pillar.

Some people have an innate flair for design: the Apple corporations and the Isambard Kingdom Brunels of this world. Some people clearly do not. Some people are the Vivienne Westwoods, Ralph Laurens, or Gianni Versaces of design – they set the fashions and open up whole new worlds of design. Others are not so original and will imitate another's style, but can still come up with novel designs within those constraints. And others shamelessly copy: less haute couture, more Top Shop.

In the realm of software design, as in many other spheres of design, designers require flair and elegance. Often we have to solve complex problems with harsh, competing forces. We aim to craft the most elegant, and least complex solution. Sometimes the designed solution is necessarily complex. It's hard to come up with a very simple solution to a complex problem.

Sadly, though, a lot of software out there is a complex solution looking for a problem to solve. I think you know what I mean. It takes real design flair to create a simple solution to a complex problem. You know that the design is 'right' when it seems so simple and so obvious that it looks like it didn't need design at all.

In the last column [1] we looked back at the consequences of a bad design; we saw how a complex solution was crafted around a relatively simple problem. It was called The Messy Metropolis. We considered software design akin to town planning (and in many ways that is a quite logical extension of the software 'architecture' analogy). We saw what a badly designed software conurbation looks like and how you can accrete one of your very own.

Now let's look in the other direction. Let's see what the town planners can do when they really try...

Design Town

The Design Town software project was superficially very similar to the The Messy Metropolis. It too was a consumer audio product written in C++, running on a Linux operating system. However it was built in a very different way and so the internal structure worked out very differently.

The prologue: I was involved with this project from the very start. A brand new team of capable programmers had been assembled to build it from scratch. The team was small (initially four programmers) and like the Metropolis, the team structure was flat. Fortunately there was none of the inter-personal rivalry apparent in the Metropolis project, or any vying for positions of power in the team. The members didn't know each other well beforehand, or how well they'd work together. But they were all enthused about the project and relished the challenge.

So far so good.

Linux and C++ were early decisions for the project, and that shaped the team that had been assembled. From the outset the project had clearly defined goals – a particular first product and a roadmap of future functionality that the codebase had to accommodate. This was to be a general-purpose codebase that would be applied in a number of product configurations.

The development process employed was eXtreme Programming (XP) [2] which, on the face of it, eschews design: code from the hip and don't think too far ahead. But this is a common mis-belief. XP does not discourage design; it discourages work that isn't necessary (this is the YAGNI – You Aren't Going To Need It principle). However, where up-front design is required XP requires you to do so. It also encourages rapid prototypes (known as spikes) to flesh out and prove the validity of designs. Both of these were very useful and contributed greatly to the final software design.

The YAGNI principle in particular paid off greatly, it encouraged us to design what we needed to early on, and defer the remaining decisions until later – when we had a clearer picture of the actual requirements and how best to fit them in to the system. This is an immensely powerful design approach, and quite liberating. One of the worst things you can do is design something you don't yet understand.

First steps

Early in the design process we established the main areas of functionality (these included the core audio path, content management, and user control/interface). We considered where they each fitted in the system and an initial architecture was fleshed out, including the core threading models that were required to achieve performance requirements.

The relative positions of the separate parts of the system was established in a conventional layer diagram, a simplified part of which is shown in figure 1. Although the system design was intentionally flexible and would grow 'organically' as pieces of functionality were added to the system, this initial architecture proved a solid basis for growth. Whereas the Metropolis had no overall picture and saw functionality grafted (or bodged) in wherever was 'convenient', this system had a clear model of what belonged where.

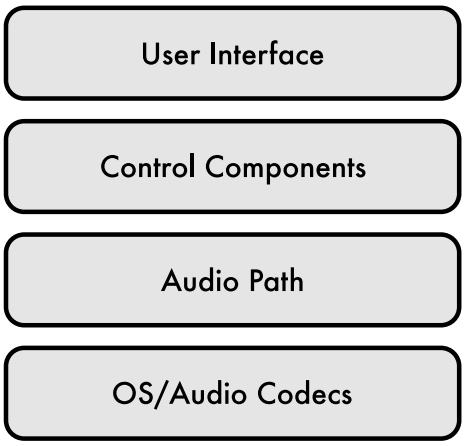


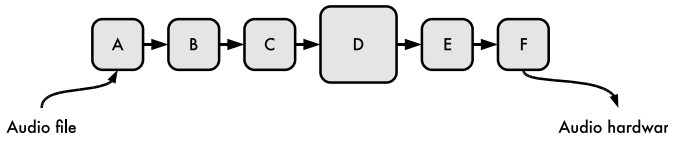
Figure 1

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

The developers believed in the design and considered it important enough to protect

Figure 2



A lot of initial design time was spent on the heart of the system: the audio path. It was essentially an internal sub-architecture of the entire system. To define this, we considered the flow of data through a series of components and arrived at a filter-and-pipeline audio architecture, similar to figure 2. The products involved a number a number of these pipelines, depending on their physical configuration.

We also made an early choice of supporting libraries the project would employ (for example, the Boost C++ libraries [3] and a set of database libraries). Decisions about some of the basic concerns were made at this point to ensure that the code would grow easily and cohesively, such as:

- the top-level code structure,
- how we'd name things,
- a 'house' presentation style,
- common coding idioms,
- the choice of unit test framework, and
- the supporting infrastructure (e.g. source control, a suitable build system and continuous integration).

These 'fine detail' factors were very important: they allied closely with the software architecture and, in turn, influenced many later design decisions.

The story unfolds

Once the initial design had been established, the Design Town project proceeded following the XP process. Design and code construction was either done in pairs (see [4] for more on pair programming) or carefully reviewed to ensure that work was correct.

The code developed and matured over time, and as the story of Design Town unfolded, these were the consequences:

- With a clear overview of the system structure in place from the very beginning, new units of functionality were consistently added to the correct functional areas of the codebase. Sometimes this was a harder job than simply bodging them into a more convenient, but less tasteful, place. So the design sometimes made developers work harder. The payoff for this extra effort was a much easier life later on, when maintaining or extending the code – there was very little cruft to trip over.
- The entire system was consistent. Every decision at every level was taken in the context of the whole design. The developers did this intentionally from the outset so all the code produced matched the design fully, and matched all the other code. Over the project's history, despite many changes ranging across the entire scope of the codebase – from individual lines of code to the architectural design – everything followed the original design template.
- The good taste and elegance of the top-level design fed down to the lower levels. Even at the lowest levels, the code was uniform and neat. This was helped by code construction techniques like pair programming, code reviews, and common code standards. However, a clearly defined software design ensured that familiar design patterns were used throughout, familiar interface idioms were adopted, and that there were no unusual object lifetimes or odd resource management issues.
- Some entirely new functional areas appeared in the 'big picture' design – storage management and an external control facility, for example. When this occurred the design, like the code, was considered malleable and refactorable [5]. One of the development team's core principles was to stay nimble – that nothing should be set in stone – and so the design, just like the code, should be changed when necessary. This encouraged us to keep our designs simple and easy to change. Consequently the code could grow rapidly and maintain a good internal structure. Accommodating these new functional blocks was not a problem.
- One of the core decisions about the codebase (which is also mandated by XP development) was that everything should be unit tested. Unit testing brings many advantages, one of which is the ability to change sections of the software without worrying about destroying everything else in the process. Some areas of the Design Town internal structure received quite radical re-work whilst the unit tests gave us confidence that the rest of the system had not been broken. For example, the thread model and inter-connection interface of the audio pipeline was changed fundamentally. This was a serious design change relatively late in the development of that subsystem, but the rest of the code interfacing with the audio path continued executing perfectly. The unit tests have us capability to change the design.

  This kind of 'major' design change slowed down as Design Town matured. After an amount of design rework, things settled down, and subsequently there were only minor design changes. The system developed quickly, in an iterative manner, with each step improving the design, until it reached a relatively stable plateaux.
- Another major benefit of the unit tests was their remarkable shaping of the code design; they practically enforced good code structure. Each small code component was crafted as a well-defined entity that could stand alone – as it had to be constructible in a unit test without requiring the rest of the system to be built up around it. Writing unit tests ensured that each module of code was internally cohesive and loosely coupled from the rest of the system. The unit tests forced careful thought about each unit's interface, and ensured that its API was meaningful and internally consistent.
- The quality control (pair programming, design and code reviews, unit tests) ensured that the system never had an incorrect, badly fitting change applied. Anything that didn't mesh with the software design was rejected.
- Design Town development was fairly pragmatic. As deadlines approached, a number of corners were cut to allow projects to ship on time. Small code 'sins' or design warts were allowed to enter the codebase either to get functionality working quickly or to avoid high-risk changes near a release. However, unlike the Messy Metropolis project, these fudges were marked as 'technical debt' and scheduled for later revision. This highlights an important attitude: the developers believed in the design, and considered it important enough to protect. They took personal responsibility for the design.
- The development timescales were neither too long nor too short (just like Goldilock's porridge). Given too long, programmers often tend to create a magnum opus (the kind of thing which will always be almost ready, but never quite materialises). A little pressure is a wonderful thing, and a sense or urgency helps to get things done. However, given too little time it simply isn't possible to achieve any worthwhile design, and you'll only get a half-baked solution rushed out. Good project planing really helps good design!

The good taste and elegance of the top-level design fed down to the lower levels.

- The development team dynamics followed the code design. Project principles mandated that no one 'owned' any area of the design, that any developer could work anywhere in the system. Everyone was expected to write high quality code, and to provide a complete suite of unit tests for their work. Whereas the Metropolis was a sprawling mess created by many uncoordinated, fighting programmers, Design Town was clean and cohesive, created by closely co-operating colleagues. In many ways, Conway's Law [6] came into effect and the team gelled together as the software did.
- The design was sufficiently well documented. There were no slavish design documents (again, this is something XP does not encourage). A good understanding of the overall code structure was provided by the design overview. Following this, the code was required to be its own documentation, both by mandating clear, simple code that clearly expresses its intent – using code structure devices (like namespaces, enumerations and classes), and clear names – but also by marking up interfaces with literate comments using the Doxygen tool [7]. Doxygen produces excellent documentation that, when used well, can reveal the code structure clearly and is accurate, too – it reflects the actual code not some out-of-date snapshot of it when a design document was last updated.
- Whilst the codebase was large, it was coherent and easily understood. New programmers could pick it up and work with it relatively easily. There were no unnecessarily complex interconnections to understand, or weird legacy code to work around.
- Since the code has generated relatively few problems, and is still enjoyable to work with, there has been very, very low turnover of team members. This is due, in part, to the developers taking ownership of the design and continually wanting to improve it.

After a length of time, the overall Design Town architecture looked something like figure 3. That is, it was remarkably similar to the original design, with a few notable changes – and a lot more experience to prove it was right. A healthy development process, a smaller, more thoughtful development team, and an appropriate focus on ensuring consistency lead to an incredibly simple, clear, and consistent design. This simplicity worked to the advantage of the Design Town, leading to malleable code, and rapidly developed products.

Where is it now?

At the time of writing, the Design Town project has been alive for three years. The codebase is still in production use and has spawned a number of successful products. It is still being developed, still growing, still being extended, and still being changed daily. It's design next month might be quite different to how it looks this month. But it probably won't – most of the major changes have already been made.

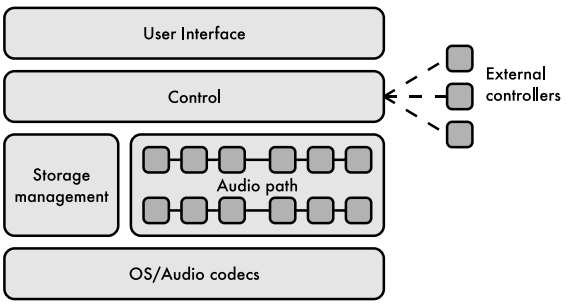


Figure 3

Let me make this clear: the code is by no means perfect, it has areas of technical debt that need work, but they stick out against the backdrop of neatness, and will be addressed in the future. Nothing is set in stone: thanks to the adaptable design and flexible code structure these things can be fixed. Almost everything is in the right place, because the design is sound.

Conclusion

Good design is the product of many factors, including (but not limited to):

- Actually doing intentional up-front design. (Many projects fail in their design at this stage)
- The quality and experience of the designers. (It helps to have made a few mistakes beforehand to point you in the right direction next time! The Metropolis project certainly taught me a thing or two.)
- Keeping the design clearly in view as development progresses.
- The team being given, and taking responsibility for the overall design of the software.
- Never being afraid of changing the design: nothing is set in stone.
- Having the right people on the team: including designers, programmers, and managers. Ensure the development team is the right size. Ensure they have healthy working relationships, as these relationship will inevitably feed into the structure of the code.
- Making design decisions at the appropriate time, when you know all the information to be able to make them. Deferring design decisions you cannot yet make.
- Good project management, with the right kind of deadlines.

So an intentional design, a good set of design decisions, and a healthy development approach can result in a vastly superior software structure. That leaves you with one less thing to worry about.

Before you turn the page and move on to the next article in this magazine stop and think about your own experience. Think about the software projects you have worked on. Which one had the worst design – and what caused it to end up like that? And which project in your past had the best design? Why? It's a good idea to reflect on your own experience and see what you can learn. (And a good moan is wonderfully cathartic.)

And just in case you were wondering, the names in these articles have been changed to protect the innocent. And the guilty. ■

Endnotes

[1] The previous column. In last C Vu. Look on the shelf. Or in the drawer. Or in the bin.

[2] XP is a lightweight, agile, development process. See http://www.extremeprogramming.org

[3] Boost is an set of excellent C++ libraries, many of which have fed into the next revision of the C++ standard library. See http://www.boost.org

[4] Pair programming is a development approach where two programmers sit together at the same computer and work on the code together. See http://www.extremeprogramming.org/rules/pair.html

[5] Refactoring is the process of improving the internal design or structure of a codebase without changing its external behaviour. Read more at Fowler's website http://www.refactoring.com

[6] Conway's Law states that team structure will follow code structure. For example, if you design a five-stage compiler, you'll create five teams to work on it.

[7] http://www.doxygen.org

Fixing Compiler Warnings the Hard Way

Thomas Guest listens when his compiler grumbles, but ignores its suggestions.

GCC makes a suggestion

The build server CC'd me on an email. Good old GCC, grumbling about operator precedence again. But Hey! – at least it had a positive suggestion to make.

```
From: buildmaster@example.com
To: lem.e.tweakit@example.com
Cc: developers@example.com
Subject: Broken build
Version: svn://svnserver/trunk@999
Platform: Linux, GCC 4.0.1
Build Log:
....
Warning: suggest parentheses around arithmetic
in operand of ^
```

I looked at the code. Listing 1 shows a simplified version, with the problem line in darker type. GCC warns:

```
$ gcc -Wall -c unpack_bits.c
unpack_bits.c: In function `unpack':
unpack_bits.c:12: warning: suggest parentheses
            around arithmetic in operand of ^
```

Setting a precedent

Needless to say, the actual offending code was buried in a longer function, indented more deeply, and with a few more indirections [1] – so it was indeed tempting to take GCC's advice and whack in a couple of brackets. Clearly the author *meant* to write:

```
bit = byte & (2^pos);
```

Why else omit spaces around the `^`?

Fortunately I live by my own rule, to avoid unnecessary parentheses, so I wasn't about to add any here without asking why. Worse than my stubborn principles, `^`, the exclusive or operator, has *lower* precedence than bitwise and, `&`, so to keep GCC happy and retain the original behaviour we'd have to write:

```
bit = (byte & 2) ^ pos;
```

This looks very bizarre code. Had it ever been exercised?

GCC was right, the code was wrong, but its diagnostic showed the wrong way to right things. On this occasion GCC should have been proscriptive, not prescriptive, and left the fix in the hands of the programmer. [2]

Don't mix bits and arithmetic

My personal rule of thumb is to avoid mixing bitwise and arithmetic operations. Although integral types support both kinds of operation, it generally feels like a type-mismatch to combine them in a single expression, or even sequence of expressions. An array of bits isn't a number, and vice-versa.

THOMAS GUEST

Thomas is an enthusiastic and experienced programmer. He has developed software for everything from embedded devices to clustered servers. His website is http://www.wordaligned.org and you can contact him at thomas.guest@gmail.com

Listing 1

```
void
unpack(unsigned char const * bits, int n_bits,
       unsigned char * buf)
{
  unsigned char bit, byte, pos;
  int b;

  for (b = 0; b != n_bits; ++b)
  {
    byte = bits[b / 8];
    pos = 7 - (b % 8);
    bit = byte & 2^pos;
    buf[b] = bit == 0 ? 0 : 255;
  }
}
```

Of course there are some treasured bit-twiddling tricks [3] which exploit the mapping between binary arithmetic and machine level register operations. So we can, for example, calculate `2` raised to the power of `19` with a simple left-shift, `1 << 19`, or test if `v` is a power of `2` with `!(v & (v - 1)) && v`. I'm not suggesting we blacklist these ingenious hacks – in fact, anyone off to an interview for a job with an embedded systems company might do well to study them. But I would say they need tucking into well-named functions.

On occasion, then, bitwise operations may legitimately be used for fast arithmetic, but the reverse, using arithmetic to pack bits, is rarely necessary. This line of code is probably wrong [4]:

```
r = h << 4 + 1;
```

The programmer probably intended the (bitwise) shift to happen before the (arithmetic) addition, like this.

```
r = (h << 4) + 1;
```

If we stick to bitwise operations, things become clear. I've written the `1` in hexadecimal as a hint it's being used as a bit pattern – sadly there's no way of writing a binary literal directly in C.

```
r = h << 4 | 0x1;
```

Anyway, the problem line in `unpack()` adheres to my rule of thumb: `&` and `^` are indeed both bitwise operations. But after some puzzling I realised the author of the code intended `2^pos` to mean `2` to the power of `pos`, *not for its arithmetic value, but for its bit pattern* – which, as every programmer knows, is a `1` followed by pos `0`s. That is, a `1` left-shifted `pos` times.

Listing 2 is what I thought the fix should be. Note, incidentally, that I've used `~0` rather than `255`, because it clearly says 'set every bit'. I'm also using unsigned integers throughout – always a good idea when working with bits.

Despite the absence of documentation, this is now at least a coherent function. It's a biterator which steps through a collection of bits (packed into bytes, the smallest memory units C offers). Each time it encounters a set/clear bit, it sets/clears all the bits in the next byte in the output buffer. That is, it expands each bit value to fill a whole byte.

This is exactly the kind of function which is surprisingly fiddly to write but simple to unit test. As already mentioned, though, the function didn't

Listing 2

```
void
unpack(unsigned char const * bits,
       unsigned n_bits, unsigned char * buf)
{
  unsigned char bit, byte;
  unsigned b, pos;

  for (b = 0; b != n_bits; ++b)
  {
    byte = bits[b / 8];
    pos = 7 - b % 8;
    bit = byte & 1 << pos;
    buf[b] = bit == 0 ? 0 : ~0;
  }
}
```

actually exist in the form shown, and the tests were all at the module level. The responsible way for me to proceed was to create a module test which exposed the defect, then make my candidate fix, confirm it did indeed fix the defect, then check the change in.

Unit test

Listing 3 shows how simple a unit test for **`unpack()`** could be. It may be longer than the function it's testing, but it's less complex. And with just a couple of test cases, it manages to cover several interesting corners of the functionality. Better still, it passes! [5]

This is white-box testing: the test knows enough about the implementation of **`unpack()`** to expose potential problems. In this case, there's something unusual about the way the **`pos`** counter goes down as the bit counter **`b`** goes up, so we make sure that the bits we're unpacking form asymmetric patterns.

Refactoring

Should we extract this tested **`unpack()`** function from its surrounding, larger, more complex function? Is it safe to do so? Have we time to spend making changes with no externally visible results? Should we tweak **`unpack()`** for efficiency (after all, it doesn't need to use the division and modulus operators each time round the loop)?

These are important questions. eXtreme Programmers refactor mercilessly [6], confident their extensive test frameworks will provide a safety net. Java programmers select the code block in their IDE then click the 'extract method' button. C and C++ programmers have less advanced tools, but Michael Feathers' *Working Effectively with Legacy Code* offers practical advice on how to transform code safely – that is, how to put it under test.

In the real world, we judge each case on merit. A nag email from the build server shouldn't necessarily trigger mass refactoring, even if the test infrastructure is in place. I think Feathers is right though, that poorly tested code is on its way to becoming legacy code: hard to adapt, unpleasant to work with, and a drag on continuing development.

Lessons

This new story repeats the same old lessons.

Set up a build server. Listen to it. Compile on multiple platforms.

Think! Compilers are concerned with syntax, not semantics. A C compiler reads your code in order to rewrite it for the machine's benefit; it doesn't understand it, that's your job.

Write small functions. Unit test them.

Integers and bit arrays are different. Be careful using bitwise operations as arithmetic shortcuts. Avoid using arithmetic for bit packing.

Oh, and in C, don't mistake **`^`** for exponentiation! ■

Listing 3

```
void
test_unpack()
{
  // Start with a varied bit-pattern.
  // Ensure each byte differs from its
  // reversed self.
  unsigned char const bits[2] =
  {
    1 << 7 | 1 << 5 | 1 << 4 | 1 << 0,
    // 10110001 binary
    1 << 6 | 1 << 5 | 1 << 3 | 1 << 0,
    // 01101001 binary
  };
  unsigned char expected[2 * 8] =
  {
    ~0, 0, ~0, ~0, 0, 0, 0, ~0,
    0, ~0, ~0, 0, ~0, 0, 0, ~0
  };
  unsigned char buf[3 * 8] = { 0 };
  unsigned char buf_copy[3 * 8] = { 0 };

  size_t const buf_size = sizeof(buf);

  // Fill the buffer with a pattern of 1s and 0s.
  // Unpack nothing and check nothing changes.
  memset(buf, 0xa5, buf_size);
  memcpy(buf_copy, buf, buf_size);
  unpack(bits, 0, buf);
  assert(memcmp(buf, buf_copy, buf_size) == 0);

  // Unpack some of the bits and check the
  // results.
  // Also check the remainder of the buffer is
  // undamaged.
  unpack(bits, 13, buf);
  assert(memcmp(buf, expected, 13) == 0);
  assert(memcmp(buf + 13, buf_copy + 13,
         buf_size - 13) == 0);
}
```

Notes and references

[1] http://c2.com/cgi/wiki?ThreeStarProgrammer

[2] I'm not complaining about GCC which did an outstanding job of flagging a genuine problem in perfectly well-defined and valid code. The other compiler frequently used on this project, MSVC V8.0, compiles this cleanly, at the same time warning standard C string functions are unsafe and *deprecated*!

[3] http://graphics.stanford.edu/~seander/bithacks.html

[4] I've taken this example directly from Andrew Koenig's *C Traps and Pitfalls*. This is a nice little book which expands on the ideas presented in a paper of the same name (http://www.literateprogramming.com/ctraps.pdf).

[5] One thing I recommend, though, is to temporarily reverse the logic in the assertions and check they then fail. Unit test frameworks often provide hooks to do this reversed-result test, which confirms the test cases are actually being run.

[6] http://www.extremeprogramming.org/rules/refactor.html

Lisp for the Web

Adam Petersen shows Lisp is still a contender.

With his essay 'Beating the Averages'[1], Paul Graham told the story of how his web start-up Viaweb outperformed its competitors by using Lisp. Lisp? Did I parse that correctly? That ancient language with all those scary parentheses? Yes, indeed! And with the goal of identifying its strengths and what they can do for us, I'll put Lisp to work developing a web application. In the process we'll find out how a 50-year-old language can be so well-suited for modern web development and yes, it's related to all those parentheses.

What to expect

Starting from scratch, we'll develop a three-tier web application. I'll show how to:

- utilize powerful open source libraries for expressing dynamic HTML and JavaScript in Lisp,
- develop a small, embedded domain specific language tailored for my application,
- extend the typical development cycle by modifying code in a running system and execute code during compilation, and
- finally, migrate from data structures in memory to persistent objects using a third-party database.

I'll do this in a live system transparent to the users of the application. Because Lisp is so high-level, I'll be able to achieve everything in just around 70 lines of code.

This article will not teach you Common Lisp (for that purpose I recommend *Practical Common Lisp* [2]). Instead, I'll give a short overview of the language and try to explain the concepts as I introduce them, just enough to follow the code. The idea is to convey a feeling of how it is to develop in Lisp rather than focusing on the details.

The Lisp story

Lisp is actually a family of languages created by John McCarthy 50 years ago. The characteristic of Lisp is that Lisp code is made out of Lisp data structures with the practical implication that it is not only natural, but also highly effective, to write programs that write programs. This feature has allowed Lisp to adapt over the years. For example, as object-oriented programming became popular, powerful object systems could be implemented in Lisp as libraries without any change to the core language. Later, the same proved to be true for aspect-oriented programming.

This idea is not only applicable to whole paradigms of programming. Its true strength lays in solving everyday problems. With Lisp, it's straightforward to build-up a domain specific language allowing us to program as close to the problem domain as our imagination allows. I'll illustrate the concept soon, but before we kick-off, let's look closer at the syntax of Lisp.

Crash course in Lisp

What Graham used for Viaweb was Common Lisp, an ANSI standardized language, which we'll use in this article too (the other main contender is Scheme, which is considered cleaner and more elegant, but with a much smaller library).

ADAM PETERSEN

Adam Petersen is a programmer and part-time psychology student. Besides spending way too much time reading tech books, Adam also has somewhat healthier hobbies like music, martial arts, modern hisotry and good literature.

Common Lisp is a high-level interactive language that may be either interpreted or compiled. You interact with Lisp through its *top-level*. The top-level is basically a prompt. On my system it looks like this:

```
CL-USER>
```

Through the top-level, we can enter expressions and see the results (user input is **highlighted**):

```
CL-USER>(+ 1 2 3)
6
```

As we see in the example, Lisp uses a prefix notation. A parenthesized expression is referred to as a *form*. When fed a form such as `(+ 1 2 3)`, Lisp generally treats the first element (`+`) as a function and the rest as arguments. The arguments are evaluated from left to right and may themselves be function calls:

```
CL-USER>(+ 1 2 (/ 6 2))
6
```

We can **de**fine our own **fun**ctions with `defun`:

```
CL-USER>(defun say-hello (to)
          (format t "Hello, ~a" to))
```

Here we're defining a function `say-hello`, taking one argument: `to`. The `format` function is used to print a greeting and resembles a `printf` on steroids. Its first argument is the output stream and here we're using `t` as a shorthand for standard output. The second argument is a string, which in our case contains an embedded directive `~a` instructing `format` to consume one argument and output it in human-readable form. We can call our function like this:

```
CL-USER>(say-hello "ACCU")
Hello, ACCU
NIL
```

The first line is the side-effect, printing `Hello, ACCU` and `NIL` is the return value from our function. By default, Common Lisp returns the value of the last expression. From here we can redefine `say-hello` to return its greeting instead:

```
CL-USER>(defun say-hello (to)
          (format nil "Hello, ~a" to))
```

With `nil` as its destination, `format` simply returns its resulting string:

```
CL-USER>(say-hello "ACCU")
"Hello, ACCU"
```

Now we've got rid of the side-effect. Programming without side-effects is in the vein of functional programming, one of the paradigms supported by Lisp. Lisp is also dynamically typed. Thus, we can feed our function a number instead:

```
CL-USER>(say-hello 42)
"Hello, 42"
```

In Lisp, functions are first-class citizens. That means, we can create them just like any other object and we can pass them as arguments to other functions. Such functions taking functions as arguments are called *higher-order functions*. One example is `mapcar`. `mapcar` takes a function as its first argument and applies it subsequently to the elements of one or more given lists:

```
CL-USER>(mapcar #'say-hello (list "ACCU"
                                  42 "Adam"))
("Hello, ACCU" "Hello, 42" "Hello, Adam")
```

The funny `#'` is just a shortcut for getting at the function object. As you see above, `mapcar` collects the result of each function call into a list, which is its return value. This return value may of course serve as argument to yet another function:

```
CL-USER>(sort (mapcar #'say-hello
                      (list "ACCU" 42 "Adam"))
              #'string-lessp)
("Hello, 42" "Hello, ACCU" "Hello, Adam")
```

Lisp itself isn't hard, although it may take some time to wrap one's mindset around the functional style of programming. As you see, Lisp expressions are best read inside-out. But the real secret to understanding Lisp syntax is to realize that the language doesn't have one; what we've been entering above are basically parse-trees, generated by compilers in other languages. And, as we'll see soon, exactly this feature makes it suitable for meta-programming.

The Brothers are history

Remember the hot gaming discussions 20 years ago? 'Giana Sisters' really was way better than 'Super Mario Bros', wasn't it? We'll delegate the question to the wise crowd by developing a web application. Our web application will allow users to add and vote for their favourite retro games. A screenshot of the end result is provided in Figure 1.

From now on, I start to persist my Lisp code in textfiles instead of just entering expressions into the top-level. Further, I define a package for my code. Packages are similar to namespaces in C++ or Java's packages and help to prevent name collisions (the main distinction is that packages in Common Lisp are first-class objects).

```
(defpackage :retro-games
  (:use :cl :cl-who :hunchentoot :parenscript))
```

The new package is named `:retro-games` and I also specify other packages that we'll use initially:

- `cl` is Common Lisp's standard package containing the whole language.
- `cl-who` [3] is a library for converting Lisp expressions into XHTML.
- `hunchentoot` [4] is a web-server, written in Common Lisp itself, and provides a toolkit for building dynamic web sites.
- `parenscript` [5] allows us to compile Lisp expressions into JavaScript. We'll use this for client-side validation.

With my package definition in place, I'll put the rest of the code inside it by switching to the `:retro-games` package:

```
in-package :retro-games)
```

Most top levels indicate the current package in their prompt. On my system the prompt now looks like this:

```
RETRO-GAMES>
```

Representing games

With the package in place, we can return to the problem. It seems to require some representation of a game and I'll choose to abstract it as a class:

```
(defclass game ()
  ((name  :initarg  :name)
   (votes :initform 0)))
```

The expression above defines the class `game` without any user-specified superclasses, hence the empty list `()` as second argument. A game has two *slots* (slots are similar to attributes or members in other languages): a `name` and the number of accumulated `votes`. To create a `game` object I invoke `make-instance` and pass it the name of the class to instantiate:

```
RETRO-GAMES>(setf many-lost-hours
                  (make-instance 'game :name "Tetris"))
#<GAME @ #x7213da32>
```

Because I specified an **init**ial **arg**ument in my definition of the `name` slot, I can pass this argument directly and initialize that slot to `"Tetris"`. The

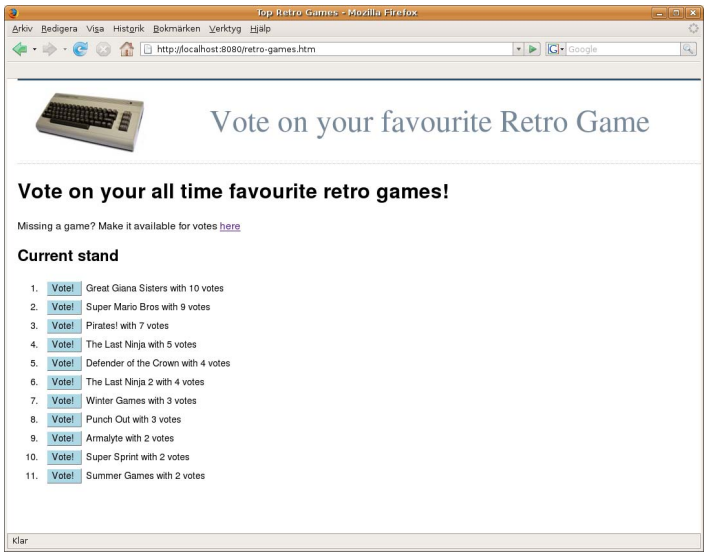


Figure 1

`votes` slot doesn't have an initial argument. Instead I specify the code I want to run during instantiation to compute its initial value through `:initform`. In this case the code is trivial, as I only want to initialize the number of votes to zero. Further, I use `setf` to assign the object created by `make-instance` to the variable `many-lost-hours`.

Now that we got an instance of `game` we would like to do something with it. We could of course write code ourselves to access the slots. However, there's a more lispy way; `defclass` provides the possibility to automatically generate accessor functions for our slots:

```
(defclass game ()
  ((name :reader   name
         :initarg :name)
   (votes :accessor votes
          :initform 0)))
```

The option `:reader` in the `name` slot will automatically create a read function and the option `:accessor` used for the `votes` slot will create both read and write functions. Lisp is pleasantly uniform in its syntax and these generated functions are invoked just like any other function:

```
RETRO-GAMES>(name many-lost-hours)
"Tetris"
RETRO-GAMES>(votes many-lost-hours)
0
RETRO-GAMES>(incf (votes many-lost-hours))
1
RETRO-GAMES>(votes many-lost-hours)
1
```

The only new function here is `incf`, which when given one argument increases its value by one. We can encapsulate this mechanism in a method used to vote for the given game:

```
(defmethod vote-for (user-selected-game)
  (incf (votes user-selected-game)))
```

The top-level allows us to immediately try it out and vote for Tetris:

```
RETRO-GAMES>(votes many-lost-hours)
1
RETRO-GAMES>(vote-for many-lost-hours)
2
RETRO-GAMES>(votes many-lost-hours)
2
```

A prototypic back end

Before we can jump into the joys of generating web pages, we need a back end for our application. Because Lisp makes it so easy to modify existing applications, it's common to start out really simply and let the design evolve as we learn more about the problem we're trying to solve. Thus, I'll start by using a list in memory as simple, non-persistent storage.

```
(defvar *games* '())
```

The expression above defines and initializes the global variable (actually the Lisp term is *special variable*) ***games*** to an empty list. The asterisks aren't part of the syntax; it's just a naming convention for globals. Lists may not be the most efficient data structure for all problems, but Common Lisp has great support for lists and they are easy to work with. Later we'll change to a real database and, with that in mind, I encapsulate the access to ***games*** in some small functions:

```
(defun game-from-name (name)
  (find name *games* :test #'string-equal
                     :key #'name))
```

Our first function **game-from-name** is implemented in terms of **find**. **find** takes an item and a sequence. Because we're comparing strings I tell **find** to use the function **string-equal** for comparison (remember, **#'** is a short cut to refer to a function). I also specify the key to compare. In this case, we're interested in comparing the value returned by the **name** function on each **game** object.

If there's no match **find** returns **NIL**, which evaluates to **false** in a boolean context. That means we can reuse **game-from-name** when we want to know if a game is stored in the ***games*** list. However, we want to be clear with our intent:

```
(defun game-stored? (game)
  (game-from-name (name game)))
```

As illustrated in Figure 1, we want to present the games sorted on popularity. Using Common Lisp's **sort** function this is pretty straightforward; we only have to take care, because for efficiency reasons **sort** is destructive. That is, **sort** is allowed to modify its argument. We can preserve our ***games*** list by passing a copy to **sort**. I tell **sort** to return a list sorted in descending order based on the value returned by the **votes** function invoked on each game:

```
(defun games ()
  (sort (copy-list *games*) #'> :key #'votes))
```

Let's define one more utility for actually adding games to our storage:

```
(defun add-game (name)
  (unless (game-stored? name)
    (push (make-instance 'game :name name)
          *games*)))
```

push is a modifying operation and it prepends the game instantiated by **make-instance** to the ***games*** list. Let's try it all out at the top level.

```
RETRO-GAMES>(games)
NIL
RETRO-GAMES>(add-game "Tetris")
(#<GAME @ #x71b943c2>)
RETRO-GAMES>(game-from-name "Tetris")
#<GAME @ #x71b943c2>
RETRO-GAMES>(add-game "Tetris")
NIL
RETRO-GAMES>(games)
(#<GAME @ #x71b943c2>)
RETRO-GAMES>(mapcar #'name (games))
("Tetris")
```

The values returned to the top level may not look too informative. It's basically the printed representation of a game object. Common Lisp allows us to customize how an object shall be printed, but we will not go into the details. Instead, with this prototypic back end in place, we're prepared to enter the web.

Generating HTML dynamically

The first step in designing an embedded domain specific language is to find a Lisp representation of the target language. For HTML this is really simple as both HTML and Lisp are represented in tree structures, although Lisp is less verbose. Here's an example using the **CL-WHO** library:

```
(with-html-output (*standard-output* nil
                   :indent t)
  (:html
    (:head
      (:title "Test page"))
    (:body
      (:p "CL-WHO is really easy to use"))))
```

This code will expand into the following HTML, which is output to ***standard-output***:

```
<html>
  <head>
    <title>Test page </title>
  </head>
  <body>
    <p> CL-WHO is really easy to use </p>
  </body>
</html>
```

CL-WHO also allows us to embed Lisp expressions, setting the scene for dynamic web pages.

Macros: fighting the evils of code duplication

Although **CL-WHO** does provide a tighter representation than raw HTML we're still facing the potential risk of code duplication; the **html**, **head**, and **body** tags form a pattern that will recur on all pages. And it'll only get worse as we start to write strict and validating XHTML 1.0, where we have to include more tags and attributes and, of course, start every page with that funny **DOCTYPE** line.

Further, if you look at Figure 1 you'll notice that the retro games page has a header with a picture of that lovely Commodore [6] and a strap line. I want to be able to define that header once and have all my pages use it automatically.

The problem screams for a suitable abstraction and this is where Lisp differs from other languages. In Lisp, we can actually take on the role of a language designer and extend the language with our own syntax. The feature that allows this is macros. Syntactically, macros look like functions, but are entirely different beasts. Sure, just like functions, macros take arguments. The difference is that the arguments to macros are source code, because macros are used by the compiler to generate code.

Macros can be a conceptual challenge as they erase the line between compile time and runtime. What macros do is expand themselves into code that is actually compiled. During their expansion macros have access to the whole language, including other macros, and may call functions, create objects, etc.

So, let's put this amazing macro mechanism to work by defining a new syntactic construct, the **standard-page**. A **standard-page** will abstract away all XHTML boiler plate code and automatically generate the heading on each page. The macro will take two arguments. The first is the title of the page and the second the code defining the body of the actual web-page. Here's a simple usage example:

```
(standard-page (:title "Retro Games")
  (:h1 "Top Retro Games")
  (:p "We'll write the code later..."))
```

Much of the macro will be straightforward **CL-WHO** constructs. Using the backquote syntax (the ` character), we can specify a template for the code we want to generate (Listing 1).

Within the backquoted expression we can use **,** (comma) to evaluate an argument and **,@** (comma-at) to evaluate and splice a list argument. Remember, the arguments to a macro are code. In this example the first argument title is bound to **"Retro Games"** and the second argument body contains the **:h1** and **:p** expressions wrapped-up in a list. In the macro definition, the code bound to these arguments is simply inserted on the proper places in our backquoted template code.

The power we get from macros become evident as we look at the generated code. The three lines in the usage example above expands into Listing 2

Listing 1

```
(defmacro standard-page ((&key title)
                         &body body)
 `(with-html-output-to-string
   (*standard-output* nil :prologue t :indent t)
   (:html :xmlns "http://www.w3.org/1999/xhtml"
          :xml\:lang "en"
          :lang "en"
    (:head
     (:meta :http-equiv "Content-Type"
            :content "text/html;charset=utf-8")
     (:title ,title)
     (:link :type "text/css"
            :rel "stylesheet"
            :href "/retro.css"))
     (:body
      (:div :id "header" ; Retro games header
           (:img :src "/logo.jpg"
            :alt "Commodore 64"
            :class "logo")
           (:span :class "strapline"
           "Vote on your favourite Retro Game"))
           ,@body))))
```

Listing 2

```
(WITH-HTML-OUTPUT-TO-STRING
 (*STANDARD-OUTPUT* NIL :PROLOGUE T :INDENT T)
 (:HTML :XMLNS "http://www.w3.org/1999/xhtml"
        :|XML:LANG| "en"
        :LANG "en"
  (:HEAD
   (:META :HTTP-EQUIV "Content-Type"
          :CONTENT "text/html;charset=utf-8")
   (:TITLE "Retro Games")
   (:LINK :TYPE "text/css"
          :REL "stylesheet"
          :HREF "/retro.css"))
  (:BODY
   (:DIV :ID "header"
    (:IMG :SRC "/logo.jpg"
          :ALT "Commodore 64"
          :CLASS "logo")
    (:SPAN :CLASS "strapline"
       "Vote on your favourite Retro Game"))
    (:H1 "Top Retro Games")
    (:P "We'll write the code later..."))))
```

(note that Lisp symbols are case-insensitive and thus usually presented in uppercase).

This is a big win; all this is code that we don't have to write. Now that we have a concise way to express web-pages with a uniform look, it's time to introduce Hunchentoot.

More than an opera

Named after a Zappa sci-fi opera, Edi Weitz's Hunchentoot is a full featured web-server written in Common Lisp. To launch Hunchentoot, we just invoke its start-server function:

```
RETRO-GAMES>(start-server :port 8080)
```

start-server supports several arguments, but we're only interested in specifying a port other than the default port 80. And that's it – the server's up and running. We can test it by pointing a web browser to http://localhost:8080/, which should display Hunchentoot's default page. To actually publish something, we have to provide Hunchentoot with a *handler*. In Hunchentoot all requests are dynamically dispatched to an associated handler and the framework contains several functions for defining dispatchers. The code below creates a dispatcher and adds it to Hunchentoot's dispatch table:

```
(push (create-prefix-dispatcher
       "/retro-games.htm" 'retro-games)
      *dispatch-table*)
```

The dispatcher will invoke the function, **retro-games**, whenever an URI request starts with `/retro-games.htm`. Now we just have to define the **retro-games** function that generates the HTML:

```
(defun retro-games ()
  (standard-page (:title "Retro Games")
    (:h1 "Top Retro Games")
    (:p "We'll write the code later...")))
```

That's it – the retro games page is online. But I wouldn't be quick to celebrate; while we took care to abstract away repetitive patterns in **standard-page**, we've just run into another more subtle form of duplication. The problem is that every time we want to create a new page we have to explicitly create a dispatcher for our handle. Wouldn't it be nice if Lisp could do that automatically for us? Basically I want to define a function like this:

```
(define-url-fn (retro-games)
  (standard-page (:title "Retro Games")
    (:h1 "Top Retro Games")
    (:p "We'll write the code later...")))
```

and have Lisp to create a handler, associate it with a dispatcher and put it in the dispatch table as I compile the code. Guess what, using macros the syntax is ours. All we have to do is reformulate our wishes in a **defmacro**:

```
(defmacro define-url-fn ((name) &body body)
 `(progn
    (defun ,name ()
      ,@body)
    (push (create-prefix-dispatcher
     ,(format nil "/~(~a~).htm" name) ',name)
     *dispatch-table*)))
```

Now our 'wish code' above actually compiles and generates the following Lisp code (macro arguments **highlighted**):

```
(PROGN
 (DEFUN RETRO-GAMES ()
  (STANDARD-PAGE (:TITLE "Retro Games")
   (:H1 "Top Retro Games")
   (:P "We'll write the code later...")))
  (PUSH (CREATE-PREFIX-DISPATCHER
          "/retro-games.htm" 'RETRO-GAMES)
        *DISPATCH-TABLE*))
```

There are a few interesting things about this macro:

1. It illustrates that macros can take other macros as arguments. The Lisp compiler will continue to expand the macros and **standard-page** will be expanded too, writing even more code for us.
2. Macros may execute code as they expand. The prefix string "/retro-games.htm" is assembled with **format** during macro expansion time. By using **comma**, I evaluate the form and there's no trace of it in the generated code – just the resulting string.
3. A macro must expand into a single form, but we actually need two forms; a function definition and the code for creating a dispatcher. **progn** solves this problem by wrapping the forms in a single form and then evaluating them in order.

Putting it together

Phew, that was a lot of Lisp in a short time. But using the abstractions we've created, we're able to throw together the application in no time. Let's code out the main page as it looks in Figure 1 (see Listing 3).

Here we utilize our freshly developed embedded domain specific language for defining URL functions (**define-url-fn**) and creating **standard-page**s. The following lines are straightforward XHTML generation, including a link to `new-game.htm`; a page we haven't specified yet. We

Listing 3

```
(define-url-fn (retro-games)
 (standard-page (:title "Top Retro Games")
  (:h1 "Vote on your all time favourite retro
       games!")
  (:p "Missing a game? Make it available for
      votes "
      (:a :href "new-game.htm" "here"))
  (:h2 "Current stand")
  (:div :id "chart" ; For CSS styling of links
   (:ol
    (dolist (game (games))
     (htm
      (:li
       (:a :href (format nil "vote.htm?name=~a"
                   (name game))
           "Vote!")
       (fmt "~A with ~d votes"
            (name game)
            (votes game)))))))))
```

Listing 4

```
(define-url-fn (new-game)
 (standard-page (:title "Add a new game")
  (:h1 "Add a new game to the chart")
  (:form :action "/game-added.htm"
         :method "post"
   (:p "What is the name of the game?" (:br)
    (:input :type "text"
            :name "name"
            :class "txt"))
   (:p (:input :type "submit"
        :value "Add"
        :class "btn")))))
```

Listing 5

```
(define-url-fn (game-added)
  (let ((name (parameter "name")))
    (unless (or (null name)
                (zerop (length name)))
      (add-game name))
    (redirect "/retro-games.htm")))
```

will use some CSS to style the **Vote!** links to look and feel like buttons, which is why I wrap the list in a **div**-tag.

The first embedded Lisp code is **dolist**. We use it to create each game item in the ordered HTML list. **dolist** works by iterating over a list, in this case the return value from the **games**-function, subsequently binding each element to the **game** variable. Using **format** and the access methods on the **game** object, I assemble the presentation and a destination for **Vote!**. Here's some sample HTML output from one session:

```
<div id='chart'>
  <ol>
   <li>
    <a href='vote.htm?name=Super Mario Bros'>
       Vote!</a>
       Super Mario Bros with 12 votes
   </li>
   <li>
    <a href='vote.htm?name=Last Ninja'>Vote!</a>
       Last Ninja with 11 votes
   </li>
  </ol>
</div>
```

As the user presses **Vote!** we'll get a request for `vote.htm` with the name of the game attached as a query parameter. Hunchentoot provides a **parameter** function that, just as you might expect, returns the value of the parameter named by the following string. We pass this value to our back end abstraction **game-from-name** and binds the result to a local variable with **let**:

```
(define-url-fn (vote)
  (let ((game (game-from-name
               (parameter "name"))))
    (if game
        (vote-for game))
    (redirect "/retro-games.htm")))
```

After a **vote-for** the requested game, Hunchentoot's **redirect** function takes the client to the updated chart.

Now when we're able to vote we need some games to **vote-for**. In the code for the retro-games page above, I included a link to `new-game.htm`. That page is displayed in Figure 2. Basically it contains an HTML form with a text input for the game name (Listing 4).

As the user submits the form, its data is sent to `game-added.htm` (Listing 5).

The first line in our URL function should look familiar; just as in our **vote** function, we fetch the value of the name parameter and binds it to a local variable (**name**). Here we have to guard against an empty name. After all, there's nothing forcing the user to write anything into the field before submitting the form (we'll see in a minute how to add client-side validation). If we get a valid name, we add it to our database through the **add-game** function.

Expressing JavaScript in Lisp

Say we want to ensure that the user at least typed something before submitting the form. Can we do that in Lisp? Yes, actually. We can write Lisp code that compiles into JavaScript and we use the ParenScript library for the task.

Unobtrusive JavaScript is an important design principle and ParenScript supports that too. But in Lisp this becomes less of an issue; I'm not actually writing JavaScript, everything is Lisp. Thus I embed my event handler in the form:

```
(:form :action "/game-added.htm" :method "post"
  :onsubmit
     (ps-inline
        (when (= name.value "")
          (alert "Please enter a name.")
          (return false)))
```

This code will compile into the following mixture of HTML and JavaScript:

```
<form action='/game-added.htm' method='post'
  onsubmit='javascript:if (name.value == "") {
    alert("Please enter a name.");
    return false;
  }'>
```

Persistent objects

Initially we kind of ducked the problem with persistence. To get things up and running as quickly as possible, we used a simple list in memory as

Figure 2

'database'. That's fine for prototyping but we still want to persist all added games in case we shutdown the server. Further, there are some potential threading issues with the current design. Hunchentoot is multithreaded and requests may come in different threads. We can solve all that by migrating to a thread-safe database. And with Lisp, design decisions like that are only a macro away; please meet Elephant!

Elephant [7] is a wickedly smart persistent object protocol and database. To actually store things on disk, Elephant supports several back ends such as PostgreSQL and SQLite. In this example I'll use Berkeley DB, simply because it has the best performance with Elephant.

The first step is to open a store controller, which serves as a bridge between Lisp and the back end:

```
(open-store '(:BDB "/home/adam/temp/gamedb/"))
```

Here I just specify that we're using Berkely DB (`:BDB`) and give a directory for the database files. Now, let's make some persistent objects. Have a look at our current `game` class again:

```
(defclass game ()
  ((name :reader   name
         :initarg :name)
   (votes :accessor votes
          :initform 0)))
```

Elephant provides a convenient `defpclass` macro that creates persistent classes. The `defpclass` usage looks very similar to Common Lisp's `defclass`, but it adds some new features; we'll use `:index` to specify that we want our slots to be retrievable by slot values. I also add an **init**ial **arg**ument to `votes`, which I use later when transforming our old games into this persistent class:

```
(defpclass persistent-game ()
  ((name :reader name
         :initarg :name
         :index t)
   (votes :accessor votes
          :initarg :votes
          :initform 0
          :index t)))
```

The Elephant abstraction is really clean; persistent objects are created just like any other object:

```
RETRO-GAMES>(make-instance 'persistent-game
                           :name "Winter Games")
#<PERSISTENT-GAME oid:100>
```

Elephant comes with a set of functions for easy retrieval. If we want all instances of our persistent-game class, it's a simple as this:

```
RETRO-GAMES>(get-instances-by-class
             'persistent-game)
(#<PERSISTENT-GAME oid:100>)
```

We can of course keep a reference to the returned list or, because we know we just instantiated a `persistent-game`, call a method on it directly:

```
RETRO-GAMES>(name (first (get-instances-by-class
                          'persistent-game)))
"Winter Games"
```

We took care earlier to encapsulate the access to the back end and that pays off now. We just have to change those functions to use the Elephant API instead of working with our `*games*` list. The query functions are quite simple; because we indexed our `name` slot, we can use `get-instance-by-value` to get the matching persistent object:

```
(defun game-from-name (name)
  (get-instance-by-value 'persistent-game
                         'name name))
```

Just like our initial implementation using `find`, `get-instance-by-value` returns `NIL` in case no object with the given name is stored. That means that we can keep `game-stored?` exactly as it is without any changes. But what about adding a new game? Well, we no longer need to maintain any references to the created objects. The database does that for us. But, we have to change `add-game` to make an instance of `persistent-game` instead of our old `game` class. And even though Elephant is thread-safe we have to ensure that the transactions are atomic. Elephant provides a nice `with-transaction` macro to solve this problem:

```
(defun add-game (name)
  (with-transaction ()
    (unless (game-stored? name)
      (make-instance 'persistent-game
                     :name name))))
```

Just one final change before we can compile and deploy our new back end: the `games` function responsible for returning a list of all games sorted on popularity;

```
(defun games ()
  (nreverse (get-instances-by-range
             'persistent-game 'votes nil nil)))
```

`votes` is an indexed slot, so we can use `get-instances-by-range` to retrieve a sorted list. The last two arguments are both nil, which will retrieve all stored games. The returned list will be sorted from lowest score to highest, so I apply `nreverse` to reverse the list (the `n` in `nreverse` indicates that it is a destructive function).

Remembering the games

Obviously we want to keep all previously added games. After all, users shouldn't suffer because we decide to change the implementation. So, how do we transform existing games into persistent objects? The simplest way is to map over the `*games*` list and instantiate a `persistent-game` with the same slot values as the old games:

```
RETRO-GAMES>(mapcar
             #'(lambda (old-game)
                 (make-instance 'persistent-game
                                :name (name old-game)
                                :votes (votes old-game)))
             *games*)
```

We could have defined a function for this task using `defun` but, because it is a one-off operation, I go with an anonymous function aka *lambda* function (see the highlighted code above). And that's it – all games have been moved into a persistent database. We can now set `*games*` to `NIL` (effectively making all old games available for garbage collection) and even make the `*games*` symbol history by removing it from the package:

```
RETRO-GAMES> (setf *games* nil)
NIL
RETRO-GAMES> (unintern '*games*)
T
```

Outro

This article has really just scratched the surface of what Lisp can do. Yet I hope that if you made it this far, you have seen that behind all those parenthesis there's a lot of power. With its macro system, Lisp can basically be what you want it to.

Due to the dynamic and interactive nature of Lisp it's a perfect fit for prototyping. And because Lisp programs are so easy to evolve, that prototype may end up as a full-blown product one day. ■

References

1 Paul Graham, 'Beating the Averages', http://www.paulgraham.com/avg.html
2 Peter Seibel, *Practical Common Lisp*, ISBN-13: 978-1590592397
3 CL-WHO, http://weitz.de/cl-who/
4 Hunchentoot, http://weitz.de/hunchentoot/
5 ParenScript, http://common-lisp.net/project/parenscript/
6 Commodore 64, photo by Bill Bertram
7 Elephant, http://common-lisp.net/project/elephant/
8 The source code for Retro Games, http://www.adampetersen.se/articles.htm

Operator Names Influence Operator Precedence Decisions (Part 2 of 2)

Derek Jones hopes for more volunteers in the future.

Introduction

This is the second of a two part article describing an experiment carried out during the 2007 ACCU conference, with the first part being published in the previous issue of C Vu [1]. This second part discusses the remember/recall assignment statement component of the experiment. See part 1 for a discussion of the experimental setup.

Children as young as four have been found to use categorization to direct the inferences they make about the world they live in [2], and many different studies have shown that people have an innate desire to create and use categories. By dividing items in the world into categories of things, people reduce the amount of information they need to learn [3] by effectively building an indexed data structure that enables them to lookup information on an item they may not have encountered before (by assigning an item to one or more categories and extracting information common to previously encountered items in those categories). For instance, a flying object with feathers and a beak might be assigned to the category *bird*, which suggests the information that it lays eggs and may be migratory.

Do developers make use of category information when trying to remember and recall information about a sequence of identifiers?

The memory for assignment statements experiments performed at the 2004 [4] and 2006 ACCU conferences [5] provides a format for testing the impact of categorization on some aspects of information storage and recall. The 2007 experiment used identifiers that were words belonging to the same category (e.g., names of trees), except for one word that did not belong to that category. That is, in the 2007 experiment a specific kind of semantic information was varied. The previous experiments attempted to measure subject's ability to remember assignment statement information over a short period of time when identifiers of different length or whose spoken form sounded alike were used. That is they varied the quantity and similarity of sound in an identifier (i.e., an identifier's spoken form).

The format of the task performed in this part of the experiment shares many features of the memory for assignment statements portion of the experiment performed in 2004 and 2006, and the write-ups of those experiments provide the common details omitted here.

Characteristics of human memory

Studies have found [6] that long term memory subsystems are meaning based, and meaning is the subject of the 2007 experiment. Most human languages are sound based and people have a short term memory subsystem dedicated to storing sounds, the subject of the 2004 and 2006 experiments.

Studies have found a wide range of factors that effect subject performance of memory for lists of information (see previous experiments for references).

Spotting the identifier that did not appear in the earlier list of assignment statements is a recognition problem, while remembering the value assigned in a recall problem. Studies have found that recognition and recall memory have different characteristics [6].

Ecological validity

Do sequences of categorically related identifiers occur in source code? There are a number of programming language constructs which associate one or more identifiers with each other, e.g., the fields of a structure type or the members of an enumeration. Developers are often exhorted to use meaningful identifiers and it is to be expected that sometimes a set of associated identifiers would be given names that reflected a shared degree of common meaning. The extent to which categorically related identifiers occur together in source code is not known.

Other issues involving ecological validity are discussed in the 2004 and 2006 articles.

Generating the assignment problems

The problems and associated page layout were automatically generated using a C program and various `awk` scripts to generate `troff`, which in turn generated postscript. The identifier and constant used in each assignment statement was randomly chosen from the appropriate set and the order of the assignment statements (for each problem) was also randomized. The source code of the C program and scripts is available from the experiments web page [7].

Selecting identifiers and integer constants

Many categories can be placed in a hierarchical relationship, Rosch [8] proposed three levels of abstraction. The highest level of abstraction being called the *superordinate-level* – for instance, the general category furniture. The next level down, called the *basic-level*, is the level at which most categorization is carried out – for instance, car, truck, chair, or table. The lowest level is the *subordinate-level*, denoting specific types of objects – for instance, a family car, a removal truck, a kitchen table. Rosch found that the basic-level categories had properties not shared by the other two categories; adults spontaneously name objects at this level and it is also the level that children acquire first. The categories used in this experiment were taken from the *basic-level* and *subordinate-level*.

The first requirement for a subject to make use of category information is that they be able to recognise that a set of identifiers belong to a category. Thus easily recognised, unique, categories are needed.

Sources for the categories selected were the top level of the Open Directory project (http://www.dmoz.org/) and Wikipedia categories (http://en.wikipedia.org/wiki/Help:Category).

A sufficient number of sets of identifiers were used that subjects would rarely encounter the same sequence. In all 20 different categories were chosen and three representative (as decided by your author) words appearing within each category (in the two sources listed above) selected; see Figure 1. This meant that the same identifiers would start to repeat after every set of 20 problems.

Observation of the category/words list after it had been used found some potential overlap between members of some categories.

- The names of trees (e.g., oak, chestnut, elm) is sometimes shared with the name of the fruit they bear (e.g., apple, pear, banana).

DEREK JONES

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

- The names of countries (e.g., france, germany, sweden) sometimes has a close association with the language spoken by natives of that country (e.g., english, dutch, mandarin).

The following is the list of 23 words, not belonging to any of the categories, used as the *not seen* identifier in the recall list.

atom	chapter	comb	engine	exterior
fence	grass	hair	hot	kettle
lizard	membrane	occult	pencil	pancil
petrol	plastic	propeller	report	room
saddle	snake	string	tail	tangent
wax	wheel	wood		

Assignment problems were created in groups of 20. Each group of 20 used one of the rows of identifiers belonging to one of the categories. The identifiers used in each assignment problem were selected by randomly choosing a row that had not already been used for the current group of 20. The recall list contained an additional identifier (the *not seen* identifier) that was not a member of the category (see second list above).

The impact of word categories is the primary concern and we want to maximise the impact of differences due to this factor. This means minimising the impact of other kinds of information (mostly integer constants) on subject performance. A good approximation to short term memory requirements is the number of syllables contained in the spoken form of the information. Choosing single digit integer constants containing a single syllable minimises their impact on short term memory load.

The integer constants were selected using the same algorithm used to generate them for the 2006 experiment.

Threats to validity

An experiment that uses semantics as the control variable depends on subjects recognizing the appropriate semantic content in the problem being answered. A failure to find a semantic effect in the results may be a consequence of subjects not recognizing the semantics rather than their failure to make use of this information.

When asked to list the strategies they used one subject listed a strategy that suggested they had noticed the semantic similarity between the words in the identifiers used. The experiment did not include any mechanism to find out whether other subjects had noticed and used category information.

Other threats to validity are discussed in the write-ups of the 2004 and 2006 experiments [4,5].

Results

Unfortunately the small number of subjects (six) who took part in the experiment was not sufficient to produce enough data to draw any statistically significant conclusions from the results. The following provides a summary of the headline results.

The average professional experience of the subjects was 14.5 years.

The answers for two subjects (i.e., 33% of all subjects) showed a close to 100% of *would refer back*.

It was hoped that at least 30 people (on the day, 6; in 2006, 18) would volunteer to take part in the experiment and it was estimated that each subject would be able to answer 20 problem sets (on the day, 23.2) in 20–30 minutes (on the day, 20 minutes).

A total of 559 answers to individual assignments were given. The average number of individual answers per subject was 93.2 with standard deviation 28.1 (95.3 in 2006, sd 38.8), the average percentage of answers where the subject would refer back was 36% with sd 47.5 (26.3% in 2006, sd 26.7), and the average percentage of incorrect answers was 6.8% with sd 7.5 (8.9% in 2006 sd 9.5).

The average amount of time taken to answer a complete problem was 51.7 (50.4 in 2006) seconds. No information is available on the amount of time invested in trying to remember information, answering the parenthesis sub-problem, and then thinking about the answer to the assignment sub-problem (i.e., the effort break down for individual components of the problem).

Figure 1

blue	red	green
chair	table	sofa
france	germany	sweden
venus	mars	jupiter
cow	sheep	pig
fly	wasp	bee
robin	blackbird	sparrow
apple	pear	banana
second	hour	minute
oak	chestnut	elm
january	june	october
heart	lung	liver
poker	scrabble	solitaire
english	dutch	mandarin
noun	adjective	verb
shirt	trousers	dress
spoon	knife	fork
river	stream	canal
plumber	painter	builder
hammer	saw	screwdriver

The raw results for each subject are available on the 2007 experiment's web page [7] (they are in the file `results.ans`; information on subject experience has been removed to help maintain subject anonymity).

Subject strategies

Discussions with subjects who took part in the 2004 experiment uncovered that they had used a variety of strategies to remember information in the assignment problem. The analysis of the threats to validity in that experiment discussed the question of whether subjects traded off effort on the filler task in order to perform better on the assignment problem, or carried out some other conscious combination of effort allocation between the subproblems. To learn about strategies used during this experiment, after 'time' was called on problem answering, subjects were asked to list any strategies they had used (a sheet inside the back page of the handout had been formatted for this purpose).

An experiment that uses semantics as the control variable depends on subjects recognizing the appropriate semantic content

The responses given to the strategies question generally contained a few sentences. Four of the six responses mentioned both the assignment and precedence problem.

The strategies listed consisted of a variety of the techniques people often use for remembering lists of names or numbers. For instance, sorting the sequence presented into a regular pattern (e.g., alphabetical) and inventing short stories involving the words and numbers.

From the replies given it was not possible to work out if subjects give equal weight to answering both parts of the problem, or had a preference to answering one part of the problem.

Evolving the Java Language: Open Source and Open Standardisation

Peter Pilgrim discusses the community process.

I am writing this article during the recent QCon Conference in London. Please accept my apologies, if you have been waiting a long time for the third article in the series.

Last year, I reported on the news from the JavaOne conference. Sun announced it was returning to innovating on the desktop, especially with its new rich media language, JavaFX. This year, it is clear that many experienced folks are diversifying away from Java. Some people are migrating away from the Java language and platform to an alternative technology like Ruby, because of the Ruby-on-Rails phenomenon. Others are still living technically on the Java Virtual Machine (JVM) platform, because they prefer to code in Scala, Groovy, Jython or JRuby. What is happening here? Well, in my personal opinion, a lot of the migration is just curiosity, developers wanting to learn a new exciting language, rumours and news stories. A lot of the time, it is demanding real-world problems that cause good engineers to look beyond Java. Other languages like Groovy and Ruby, of course, already offer closures and dynamic types (so called duck typing: if it walks like duck, quacks like a duck, then it probably is a duck). A language like Scala can support functional programming and also the actor model for concurrency.

if it walks like duck, quacks like a duck, then it probably is a duck

As a Java Champion, I felt privileged to be on a recent conference call to hear James Gosling, vice president and creator of Java, talk about the 'Feel of Java, Revisited'. In the lecture at Sun's Santa Clara Campus auditorium, he described the early foundation of the Green project that would eventually become Java. The philosophy of language was originally a programming language designed for blue-collar workers. Gosling described the dichotomy his team faced bridging the gap between designing an easy-of-use scripting language (e.g. like the then AppleScript) versus hard-core computer science compiler language type (e.g. Lisp / C / C++ / SmallTalk).

It was rather amusing and ironic, now, that Mr. Gosling used the term scripting language in his presentation in California. If you think back to 1995, then not having to deal with system header files and object linkage was rather decent. Add to the fact, there was a portable virtual machine that could interpret byte-codes, a built-in garbage collector and a set of core library APIs available by default for any supporting operating system with a runtime-environment. There was also rudimentary networking over sockets, again built-in, being embeddable in a HTML page, and some semblance of security then it was all then pretty revolutionary. It is clear that Java was winner, and Gosling even quipped, 'Getting bankers to use G.C. Wow! One of the achievements that I am most proud of'.

So what has caused the migration away from Java? What is the reason that engineers have decided to move beyond Java in recent times? The platform was very successful for enterprises and corporations, especially on the server-side and web applications. There was innovation happening all around with software islands and third party repositories, examples such as the Apache Software Foundation, which was open sourced. There were early interest in server side and mind-share on development. Java was and

PETER PILGRIM

Peter is a Java EE software developer, architect, and Sun Java Champion from London. By days he works as an independent contractor in the investment banking sector. Peter can be contacted at peter.pilgrim@gmail.com

Operator Names Influence Operator Precedence Decisions (continued)

Conclusion

Because of the small number of subjects who took part in the experiment is not possible to draw any statistically significant conclusions from the results (although running the experiment on a Friday seems to be a poor idea).

Further reading

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

Acknowledgments

The author wishes to thank everybody who volunteered their time to take part in the experiment and those involved in organising the ACCU conference for making a conference slot available in which to run it. ■

References

1 D.M. Jones, 'Operand Names Influence Operator Precedence Decisions', *C Vu*, 20:1 pp5–11, Feb 2008.

2 S. A. Gelman and E. M. Markman, 'Categories and induction in young children', *Cognition*, 23:183–209, 1986.

3 E. M. Pothos and N. Chater, 'Rational categories' in *Proceedings of the Twentieth Annual Conference of the Cognitive Science Society*, pp 848–853, 1998.

4 D. M. Jones, 'Experimental data and scripts for short sequence of assignment statements study', http://www.knosof.co.uk/cbook/accu04.html, 2004.

5 D. M. Jones, 'Experimental data and scripts for developer beliefs about binary operator precedence', http://www.knosof.co.uk/cbook/accu06.html, 2006.

6 J. R. Anderson, *Learning and Memory*, John Wiley & Sons, Inc, second edition, 2000.

7 D. M. Jones, 'Experimental data and scripts for operand names influence operator precedence decisions', http://www.knosof.co.uk/cbook/accu07.html, 2008.

8 E. Rosch, C. B. Mervis, W. D. Gray, D. M. Johnson and P Boyes-Braem, 'Basic objects in natural categories', *Cognitive Psychology*, 8:382–439, 1976.

is popular today with enterprise computing. Businesses trusted the platform to run critical operations and vendors created products that ran on or with Java. There was one crucial arbiter here, a standardisation body was created in 1997. It was called the Java Community Process.

The Java Community Process is the international body that standardises new APIs and extra functionality for the Java platform. JCP consists of at least three Executive Committee groups (ECs), which are composed of representatives from Sun and non-Sun companies. EC also includes an individuals or two, who for the most part are independent of any company. The JCP manages hundreds of Java Specification Requests (JSRs). Each JSR is managed by specification lead (spec-lead). There can more than one spec-lead and some are individuals independent from any company. Anybody, can submit a JSR, which is approved or rejected by one of the Executive Committees. JSRs take a long time to be completed, because there is a lot of documentation, protocol and design and requirement to build a reference implementation and test kit for standardisation. There are JSRs for almost everything you can think of in Java, for example JSR-275 is about adding a Measurements and Units API as standard to the Java platform.

Here is the problem: The JCP was not open enough in the earliest days of Java, circa 1998, it was dominated by employees from very powerful companies, obviously Sun Microsystems, but also including IBM, BEA, Oracle, HP and others. Individuals and representatives of open source project have found it very difficult to influence genuine change in Java. Because corporations have had their own vested interest in their customers and profit lines, they have politically not supported or shunned technologies that were often de-facto, popular or open source. Most of time these libraries existed outside the JCP body. This has lead to detriment of support for the JCP, and developers voting with their feet to use de-facto technologies or follow movements. For instance there were a progression to not use earlier EJB specifications and associated application servers. Other engineers innovated with lightweight equivalents such as Spring Framework and Hibernate for their operational requirements.

So let me bring this story up to date: I am sitting in the QCon conference where I was invited to participate on a panel discussion of the JCP and its association with open source and openness. The current chair person, Patrick Curran, who is a Brit living in Silicon Valley, seems very keen to introduce change into the standards body. There were a lot ideas that were exchanged in our session and also at our BOF later in the evening. I suggested some of my own ideas: make the JCP easier for ordinary developers and engineers to join. I borrowed one idea that came from ACCU, perhaps encourage users group through companies to sponsor standardisation meetings for some of the JSRs. (ACCU supports the ISO C++ committee with meetings in London.)

The JCP, in my humble opinion, has to be open going forward or else other companies may decide to form there own standards committee. The JCP already conflicts with the OSGi Alliance for modularisation and dynamic extensions JSR 277 and 294. Some corporations have already decided to do their own thing, such as Google calling its similar language, a Dalvik executable and neatly circumventing the licensing costs for Java Micro-Edition for every single popular mobile phone in the market. (See the Android platform for further details on that.) With the Android platform, I think that a warning shot has probably already been fired across the bows of the JCP frigate, that a group of corporations could get together and decide to develop a successor to Java independently from Sun. With that they can add an implementation of Closures, Control Abstractions or any other extensions they can dream up. Unfortunately they cannot call it Java, because of licensing and intellectual property rights issues, but I fear that this could have a serious split of the future Java platform, if there is not enough openness in the Java Community Process.

For the record, there is currently a lot of debate on closure syntax for Java. I think many Java developers can understand the benefits of anonymous functions and closures when they are described sufficiently or experienced in other languages that already have them. There are three prominent Java closure specifications (BGGA, FCM and CICE). However, closures are radical department from C++ blue collar language of 1995. There are many corporations who would love to see it in the JDK 7 or maybe 8, but until Sun and Google decide to commit real human resources to it, then it is hard to see this feature appearing any time soon on the Java platform.

Finally, we have reach end of yet another world view. I look forward to meeting you at the ACCU Conference 2008 in person.

Postscript: The one thing that Sun does have in a favour is the Java Virtual Machine. Most Java experts consider the JVM the crown jewels of the entire platform. Sun has hired JRuby developers and has this year recruited major Python developers , Tim Leung and Frank Wierzbicki (Python on the Java VM project). This is probably, because Sun also bought MySQL earlier this year. ■

References

Here are the URLs:

http://openjfx.org/
https://java-champions.dev.java.net
http://www.jcp.org/ (Java Community Process)
http://www.jcp.org/en/procedures/jcp2 (JCP Procedures)
http://www.jcp.org/en/jsr/detail?id=275 (Measurements and Units Specification 3.1)
http://www.jcp.org/en/jsr/detail?id=318 (Enterprise Java Beans 3.1)
http://weblogs.java.net/blog/cayhorstmann/archive/2008/03/feel_of_java_re.html (Feel of Java, Revisited)
http://jaoo.dk/london-2008/conference/
http://jaoo.dk/london-2008/presentation/Panel%3A+Open+Source+and+Open+Standards
http://www.javac.info/ (Closures for the Java language, BGGA)
http://www.jroller.com/scolebourne/entry/fcm_prototype_available (First Class Methods closures proposal)
http://jruby.codehaus.org/
http://www.pythonthreads.com/news/latest/sun-invests-into-python-scripting---hires-jython-developers.html
http://fwierzbicki.blogspot.com/2008/02/jythons-future-looking-sunny.html (Frank Wierzbicki Blog)
http://www.jython.org/ (Jython project)

Future-Proofing your Python Scripts

Silas Brown keeps your scripts working.

This article briefly outlines how to make your Python code ready for Python 3 while still able to work on Python 2. As Python 3 is not yet complete, this cannot be authoritative, but I hope these suggestions are helpful.

- Avoid the **print** statement, because it's going to become a function, so there's no way to write code that will work in both Python 2.x and Python 3 (unless you want to introduce multiple versions of your code).

  You can use **sys.stdout.write()** instead, and remember to include the newline. (But if you're using print for logging then consider writing to **sys.stderr**, or using the 'logging' library that has been present since Python 2.3.)

- **dict.keys()**, **dict.values()** and **dict.items()** will return 'views' (generators) instead of lists. If you must have lists then you can still get them by saying **list(dict.keys())**, and the code shouldn't run much slower than it does already in either Python 2 or 3. **dict.iterkeys()** etc should be replaced by **dict.keys()** to work in both versions, albeit at the expense of a slow-down in version 2.

  (Note that in Python 2.4 and up you don't need to convert a dictionary to a list just to sort it, because you can use the **sorted()** built-in function and that will continue to work in Python 3. But there are still a lot of current Mac OS X machines out there whose Python version is 2.3, so I still like to be Python-2.3 compatible if possible.)

- The same goes for **range()** (**xrange** will go), and **zip()** (**iterzip** will go). But changing all your existing **xrange()** calls into **range()** calls could give quite a performance hit in Python 2.x, so I suggest doing the following at the start of your code:

```
# map xrange to range if it doesn't exist
#(Python 3)
try: xrange
except: xrange = range
```

- **file.xreadlines** will be called **file.readlines**. But changing existing code to say **file.readlines()** could cause problems in Python 2.x if the file is very large or is piped output from a program.

  If they don't keep **xreadlines** as a backward-compatibility alias, then you might need to do something like this:

```
def xreadlines(fileObj):
  func = getattr(fileObj, 'xreadlines',
     fileObj.readlines)
  return func()
  ...
for line in xreadlines(
   popen("some-command")): ...
```

  which should work, because you can return an iterator.

- **dict.has_key** will go. Instead of writing **dict.has_key(5)**, you need to write **5 in dict** – and you can start doing this in Python 2.x.

- Make sure you're not using **<>** as an alias for **!=**, because it will be dropped

- There will be no more classic classes. I don't know whether the syntax **class X:** will be modified to make a new-style class, or whether it will raise an exception. However, 'new-style' classes have been around since Python 2.2, so you can start using new-style classes now. Simply change **class X:** to **class X(object):** and you should be safe. (The new-style classes also let you do other things like static methods and computed properties; see the documentation for details.)

- Support for string exceptions will be removed. If you use **raise "message"**, change it to **raise Exception("message")** (or even better, define your own class). This will work in Python 2.x and 3.

- If you do have your own exception types, in Python 3 they must inherit from **BaseException**, which doesn't exist in all 2.x versions. However, the class **Exception** does exist in 2.x versions and will inherit from **BaseException** in Python 3, so if you now make sure all your exception types inherit from **Exception**, you should be OK.

- Dividing an **int** by an **int** will return a **float**. This could break code that assumes it will be rounded down to an integer.

  You need to check all your divisions and see if you need to add **int()** around any of them (or consider using shift operators if you're dividing by a power of 2, which will give a speed advantage if the program is to be run on machines with emulated floating point).

- **sys.exc_type** etc will go, but **sys.exc_info()** will stay so switch to using that.

- Other things to watch out for: **apply**, **buffer**, **coerce** and **input** will all need to be re-written in other ways as they will be deleted. (You can still use **raw_input**.)

It's worth doing the above, especially if you distribute scripts with

```
#!/usr/bin/env python
```

at the top – in the future you will have no idea whether that will invoke Python 2 or Python 3. ■

SILAS BROWN

Silas is partially sighted and is currently undertaking freelance work assisting the tuition of computer science at Cambridge University, where he enjoys the diverse international community and its cultural activities. Silas can be contacted at ssb22@cam.ac.uk

Storm in a Teacup

Tim Penhey introduces the wonders of Storm.

Storm[1] is an object-relational mapper (ORM) written in Python. The purpose of an ORM is to provide a simple interface between an object oriented programming language, python in this case, and a relational database. One thing that Storm is not is a tool that generates the SQL schema for you from class definitions.

The Storm website[1] lists the following design benefits:

- Clean and lightweight API offers a short learning curve and long-term maintainability
- Storm is developed in a test-driven manner. An untested line of code is considered a bug
- Storm needs no special class constructors, nor imperative base classes
- Storm is well designed (different classes have very clear boundaries, with small and clean public APIs)
- Designed from day one to work both with thin relational databases, such as SQLite, and big iron systems like PostgreSQL and MySQL
- Storm is easy to debug, since its code is written with a KISS principle, and thus is easy to understand
- Designed from day one to work both at the low end, with trivial small databases, and the high end, with applications accessing billion row tables and committing to multiple database back-ends
- It's very easy to write and support back-ends for Storm (current back-ends have around 100 lines of code).

And the following features:

- Storm is fast
- Storm lets you efficiently access and update large datasets by allowing you to formulate complex queries spanning multiple tables using Python
- Storm allows you to fall-back to SQL if needed (or if you just prefer), allowing you to mix 'old school' code and ORM code
- Storm handles composed primary keys with ease (no need for surrogate keys)
- Storm doesn't do schema management, and as a result you're free to manage the schema as wanted, and creating classes that work with Storm is clean and simple.
- Storm works very well connecting to several databases and using the same Python types (or different ones) with all of them
- Storm can handle **`obj.attr = <A SQL expression>`** assignments, when that's really needed (the expression is executed at **`INSERT`**/**`UPDATE`** time)
- Storm handles relationships between objects even before they were added to a database
- Storm works well with existing database schemas
- Storm will flush changes to the database automatically when needed, so that queries made affect recently modified objects.

An untested line of code is considered a bug

These are bold claims indeed. In this article I'm only going to cover some simple Storm usage, however I think that it would be great if someone else (or two) would take up a challenge to either prove or disprove some other of these claims.

Getting Storm

I am not aware of any packages at this stage to install Storm (on any operating system). Storm is available as a Bazaar[2] branch from Launchpad[3]. As long as you have a relatively up-to-date Bazaar client (1.0 or later), you can get the storm branch using:

```
bzr branch lp:storm
```

This will create a directory called `storm` in the directory that you executed this command. In this directory there are the normal files that you expect to see in open source software, such as `LICENSE`[4], `NEWS` and `README`. You will also find a directory called `storm` that is the Python module, and a python script called `test`, along with a `tests` directory. Running `./test` took around 19 seconds on my somewhat slowish VAIO laptop and executed the 1845 test cases and the 131 doctests, so I can attest to the claim that the code is tested, but I can't say what the coverage is like.

Trying out Storm

Before we go much further, it is worth checking that we can import Storm. For these examples I'm just running the python interpreter from the directory that was created when I grabbed the code. Alternatively you could add that directory to the `PYTHONPATH` environment variable.

```
>>> import storm
>>> storm.version
'0.12'
```

For a database I'm going to use an in-memory SQLite[5] database. To get the python bindings for SQLite (for Ubuntu at least) install the `python-sqlite` package.

A number of the code examples shown here are taken from the Storm tutorial[6]. We will start by defining a `Person`.

```
>>> from storm.locals import *
>>> class Person(object):
...     __storm_table__ = "person"
...     id = Int(primary=True)
...     name = Unicode()
```

Storm requires the classes to be 'new style' as it uses descriptors internally, and descriptors do not fully work with old style classes. Looking at the class definition above, it looks fairly normal except for the **`__storm_table__`** member. This is a special member variable that Storm looks for to identify the underlying table that is going to get new rows when we store the instances. I used the term *store* deliberately as a **`store`** is a core part of how the code interacts with Storm. The store is the primary interface to the database, and it manages transactions with **`commit`** and **`rollback`**, caching as well as other high-level functions such as querying with **`find`**. A **`store`** is created with a **`database`** instance, and a **`database`** is created with the **`create_database`** method. Creating an in-memory SQLite database is very simple:

```
>>> database = create_database('sqlite:')
```


TIM PENHEY

Tim is currently working for Canonical on Launchpad doing interesting things with Python, Zope and Bazaar. Tim lives in New Zealand, supports the All Blacks, and doesn't get enough exercise. Tim can be reached at tim@penhey.net

Desert Island Books

Jez Higgins plans for a long stint alone.

All of us know Jez. Either personally or as ACCU chairman. He needs little introduction, but I nagged him for a profile anyway.

> Jez works in his attic, living the on-and-off life of a journeyman programmer. He is currently teaching himself how to make balloon animals. In April 2006, he became ACCU Chair. His website is http://www.jezuk.co.uk/

I would add to that that Jez has been, and continues to be, one of the significant people in my career and I owe him much. I thoroughly enjoyed reading about his selection of books and he inspired me to buy the first book he mentions.

Paul Grenyer

What's it all about?

Desert Island Disks is one of 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.

Jez Higgins

Da-da-da-deeee-da-da-deeee!

Stranded on a desert island with a handful of books and a couple of oggs on a memory stick. It's a prospect that's both alarming and seductive. Fingers crossed for a reasonably temperate island (although I guess that might not meet the accepted definition of desert) because I'm not desperately happy in the sun. I'm English, you know.

The first two books I'd grab are *Software Tools in Pascal* by Brian Kernighan and P J Plauger [1] and Jon Bentley's *Programming Pearls* [2]. If you are spending an indeterminate length of time on this island, you need books you can read and re-read. Since I have already read and re-read both books and look forward to reading them again, I reckon they'll stand that test.

Although I'd heard of and had flicked through both books beforehand, I first encountered them properly a bit over 10 years ago. I was working for Zuken-Redac, one of those world leading companies you've never heard of. The work, on part of a PCB/MCM/EDA [3] suite, was both hard and interesting. The code was in several layers, and the further down you went the further back in time you traveled. At the top was the user interface and user scripting layer, written in a proprietary Forth variant. Next to the Forth interpreter was the heart of the application, which was based around a Smalltalk-like object system, complete with virtual message dispatch, meta-classes, and garbage collection, all written in C. That was simultaneously very clever and highly confusing. Buried in a comment in the depths of the source was a reference to the famous Smalltalk issue of Byte [4], which had apparently inspired the whole thing. Somewhere else

Storm in a Teacup (continued)

And creating a store for that database is also quite simple.

```
>>> store = Store(database)
```

We can execute arbitrary SQL using the **store**, and we'll do so to create the **person** table.

```
>>> store.execute('CREATE TABLE person '
...               '(id INTEGER PRIMARY KEY, '
...               ' name VARCHAR)')
```

This returns a result set that we can safely ignore (it actually contains a single result of no values). Now we have somewhere to put our people, let's create one.

```
>>> eric = Person()
>>> eric.name = u'Eric the Viking'
>>> print eric.id
None
```

Eric initially has no id, as you'd expect from the code. To add Eric to the database we use the **add** method on the **store**.

```
>>> store.add(eric)
<__main__.Person object at 0x82a2bac>
```

Getting Eric out of the database uses the store **find** method. The **find** method returns a **ResultSet**. A **ResultSet** has a convenience method, **one**, that extracts the sole item in the result set. Since the item being searched for is in fact Eric, Storm is smart enough to return a reference to the single instance of Eric.

```
>>> rs = store.find(Person,
                    Person.name == u'Eric the Viking')
>>> rs.one() is eric
True
>>> eric.id
1
```

Conclusion

Now that is just the briefest of tastes of Storm. There are many more things that Storm can manage for you. If you want to see more, you can wait for the next article, or get Storm yourself and have a play following the tutorial. ■

Notes and references

[1] http://storm.canonical.com

[2] http://bazaar-vcs.org

[3] https://launchpad.net

[4] Storm is licensed under the GNU LGPL 2.1

[5] http://www.sqlite.org

[6] https://storm.canonical.com/Tutorial

there was an object persistence layer which stashed things into an Informix database. That was written in C++. Right down in the bowels was a whole load of genuinely pre-ANSI C. Regardless of any opinion you might have formed of the code from this description, it was stable and worked really well. Indeed, I believe it's still in use. The code was also portable across pretty much every flavour of Unix then around, regardless of integer size or endianess. It was though, with the exception of that reference to Byte, almost entirely undocumented. Of course. I spent the first two weeks on the job writing Perl scripts to parse out the data structures that defined the Smalltalky objects, building myself a nice little inheritance diagram. After that I was able to cull out a load of unused classes left over from another application, and things became a little clearer.

So it was fun, if rather tiring. Every couple of hours, my colleague Steve (charged with porting the whole shebang to Window NT) and I would stroll the length of the corridor to the Double-Decker and coffee vending machines for a 10-second oil break. The part of the building we were in was largely deserted, and we rarely saw anyone else. One day, though, we bumped into a lady as she emerged from an adjacent door. She was, she revealed, the company librarian and behind the door was the company library, which we were welcome to use whenever we wished.

it's the kind of book you read at a particular time in your life

To be honest, most of the library's books weren't particularly scintillating: old VAX manuals, ageing electronics text books, that kind of thing. There was the odd little gem, if you looked hard enough, and that's were I first found and read these two books.

Software Tools in Pascal stands re-reading because it has, unusually for a technical book, a terrific narrative. It starts with a tiny task – copy everything from the console input to the console output – and presents the correspondingly tiny program. Step by step, program by program, you arrive at the end of the book with an ex-like line editor, a roff-style print formatter, and a macro processor. En route, you take in filtering, file archiving, sorting, and regular expressions. Each incremental step seems so logical and the code presented is so clear, that you just want to keep reading. Ordinarily, I find large chunks of code in a book rather tedious, but Kernighan and Plauger's code is a joy. The lessons it imparts on simplicity, clarity, efficiency, on tools and the Unix philosophy, in common sense, how each decision effects the finished program – well, they are at the core of what we do, and how we should think about programming.

Bentley's book is similarly stimulating. *Programming Pearls* is a collection of columns written for the ACM and so, while there are several running themes, each one stands largely by itself. In each column, Bentley presents some problem, and examines various solutions, before ending with further questions for the reader. They're not trick problems with a single definite answer hinging on some detail of operator precedence or something equally trivial. Instead, Bentley takes some field of programming, often something quite common and that you will have encountered, and picks it apart, illuminating the darker corners, revealing the core of the problem. Like Kerninghan and Plauger, Bentley delights in simplicity, elegance, clarity without ever jamming them down your throat, and he writes with intelligence and a certain wit. Having read it cover to cover, I still dip into it periodically for inspiration and it never fails energise and enthuse.

I don't know if my choosing two books written over 25 years ago says anything about me, or about the state of the programming books, or simply reflects badly on my technical library. I'm not aware, with the exception of Kernighan's other work, of similar books. I'd welcome suggestions for when I make it back to civilisation.

Philip and Alex's Guide To Web Publishing [5] is my third choice. I've built web interfaces but I'm not a web designer. I've build a number of e-commerce apps but I'd never describe myself as a web programmer. I have a website but I wouldn't say I was a web publisher. Nonetheless, I really enjoy this book. Greenspun is clever, and cocky, and funny, and he talks a lot of sense about building websites. Alex is his dog. It a good looking book too, it's probably the only technical book you can leave out on your coffee table.

I'm assuming that I was washed up on this island with some kind of computer and development environment. Perhaps a little Asustek EEE[6] with a little solar panel. Maybe a jerry-rigged Trevor Baylis-style clockwork gizmo for when it's raining. My off-hours project for the past couple of years has been writing an XSLT processor in C++[7]. If I'm going to have some serious time to spend on it, I might as well go the whole hog and have a crack at an XSLT 2.0 processor, which makes my fourth choice the XPath 2.0/XSLT 2.0/XQuery specifications. There aren't a huge number of XSLT 2.0 processors around, and so it'd be nice to come back and join that little club. There's also something stimulating yet relaxing about coding up a standard. I can't quite explain why – it's something to do with trying to implement something that can be quite rigorous and challenging, whilst secure in the knowledge that the requirement isn't going to change once you've done it.

My choice of novel is Don DeLillo's *Underworld* [8]. I bought the book a few years ago, stashing it on my pile of stuff to read. Several of my friends had copies and were raving about how good it was, although it subsequently emerged not all of them had actually read it. I kept putting it off too – its 800+ page bulk is rather daunting. Some time later, I saw my then boss had a copy on his desk. He confirmed he had read it and when I asked what it was like, he didn't reply directly saying "it's the kind of book you read at a particular time in your life". Having read it, being stuck on an island might be the time to read it again.

Pete Seeger's *Song and Playtime* [9] and Bob Mould's *Workbook* [10] are my album choices. I like to sing, but I don't have any great range and my ability to hold a tune varies on a day-by-day, sometimes hour-by-hour, basis. Seeger, of course, is a towering figure in American folk and strongly believes in the power of music and song as an agent of political and social change. He never forgets, though, that music is fun. *Song and Playtime* is a collection of children's songs. The arrangements are very simple, often simply Seeger's voice, sometimes with banjo of hand-clap accompaniment, but every one fizzles with sing-a-long energy. You just can't help but sing, or tap your foot, or dance. It's a wonderful record. Bob Mould's appeal is, I grant you, not as universal. While his name might not be familiar, it's quite like you have things in your own music collection that bear his influence. Me, I love the man dearly, and I'd take this throbbing, angry, reviving album with me wherever I was stranded. ■

References

[1] *Software Tools in Pascal* by Brian W Kernighan and P J Plauger, Addison Wesley Professional. ISBN 0-201-10342-7. Used copies selling for pence on Amazon, around £20 new.

[2] *Programming Pearls* 2nd Ed by Jon Bentley, Addison Wesley. ISBN 0-201-65788-0. Around £20.

[3] Printed circuit board/multi-chip module/electronic design automation – circut board layout and design tools.

[4] Extracts are available from http://www.byte.com/, including Larry Tesler on 'The Smalltalk Environment' at http://www.byte.com/art/9608/sec4/art3.htm.

[5] *Philip and Alex's Guide To Web Publishing* by Philip Greenspun, Morgan Kaufman. ISBN 1558605347. Available online at http://philip.greenspun.com/panda/

[6] Assuming they ever get them back in stock.

[7] Arabica, http://www.jezuk.co.uk/arabica. You might have seen me mention it before.

[8] *Underworld* by Don DeLillo, Picador. ISBN 0330369954.

[9] Pete Seegar's *Song and Playtime*, originally released in 1960 was remastered and release on CD by Folkways in 2001

[10] *Workbook* by Bob Mould, Virgin Records, 1989

Next issue: Kevlin Henney picks his desert island books.

Code Critique Competition 51

Set and collated by Roger Orr.

A book prize is awarded for the best entry.

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

Last issue's code

I'm trying to sort a C# `KeyedCollection` generic class but I find sometimes the sorting seems to hang. Can you suggest what I'm doing wrong?

There are at least two major problems with the code but, as always, try to help the writer help themselves. The code is shown in Listing 1.

Critique

There was only one critique this issue. Come on, readers, I'm sure that more than one of you could have put fingers to keyboard and supplied a critique of this code!

Listing 1

```
using System;
using System.Collections.ObjectModel;
using System.Reflection;
public class FieldCollection :
  KeyedCollection<object[],object> {
  private string[] fields;
  public FieldCollection(
    params string[] fields ) {
    this.fields = fields;
  }
  protected override object[] GetKeyForItem(
    object item ) {
    object[] keys =
      new object[ fields.Length ];
    for (int i=0; i<fields.Length; i++) {
      string field = fields[i];
      FieldInfo fieldInfo =
        item.GetType().GetField( field,
          BindingFlags.Instance |
          BindingFlags.NonPublic |
          BindingFlags.Public );
      keys[i] = fieldInfo.GetValue( item );
      }
    return keys;
  }
  public void Sort() {
    bool sorted = false;
    while (!sorted) {
      sorted = true;
      for(int i=1;i<base.Count && sorted;i++){
        Collection<object> collection = this;
        object object1 = collection[i-1];
        object object2 = collection[i];
        object[] key1= GetKeyForItem(object1);
        object[] key2= GetKeyForItem(object2);
        for (int j=0; j<fields.Length; j++) {
          IComparable key =
          key1[j] as IComparable;
          if (key != null) {
            if (key.CompareTo(key2[j]) > 0) {
              base.RemoveAt(i);
              base.Insert(i-1, object2);
              sorted = false;
              break;
            }
          }
          else {
            throw new Exception();
          }
        }
      }
    }
  }
}
public class TestColl {
  private string firstname;
  private string lastname;
  public TestColl( string name ) {
    this.firstname = name.Split(' ')[0];
    this.lastname = name.Split(' ')[1];
  }
  public override string ToString() {
    return string.Format("{0} {1}",
      firstname, lastname);
  }
  public static void Main(string[] args) {
    try {
      test();
    }
    catch (System.Exception ex) {
      System.Console.WriteLine(ex);
    }
  }
  private static void test() {
    FieldCollection coll = new
      FieldCollection(new string[]
      {"lastname", "firstname"});
    coll.Add(new TestColl("Roger Orr"));
    coll.Add(new TestColl("Alan Griffiths"));
    coll.Add(new TestColl("Tim Penhey"));
    coll.Add(new TestColl("Kevlin Henney"));
    // Doesn't sort if I add this one:
    // coll.Add(new TestColl("Jez Higgins"));
    coll.Sort();
    foreach (TestColl test in coll) {
      System.Console.WriteLine(test);
    }
  }
}
```

Listing 1 (cont'd)

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

Simon Sebright <simonsebright@hotmail.com>

Cor blimey govenor, wot kinda language is this? Usually you come with C or C++, this is a turn up for the books. Hmm, so what seems to be the trouble, then?

The code? Did you? Aha, let's have a look in the Inbox. Oh, it's in the spam folder! Anyway...

... Right, what do you want me to do?

Tell you what's wrong! You must be joking! No, you are going to tell me how to fix it, and fix it. Right, what is this code all about then?

OK, so you've got these objects you want to sort. How?

Oh, I see, you want to sort them based on the fields in the objects. What's all this `KeyedCollection` stuff, then?

Ah, so it's in MSDN. Yes, but tell me what you are doing with it.

Ah, you found a base class for object which can be stored in a dictionary, where the key for the dictionary is in the object. Clever, saves space, I suppose.

Hmm, but you are not putting these things in a dictionary.

Yes, I see, you have written a sort function. And very good, you have some test code. Super. So, what exactly is going wrong? It's hanging? Hmm, I don't think so. What have you done about it?

Lazy so-and-so. Right, let's see. How about you pop that `System.Console.WriteLine()` stuff into your `Sort()` function loop? Then you might see what is going on...

... Oh, hello, found anything with that? Ah, good, what's happening? Really, the values keep getting changed around in order? Hmm, that's not hanging is it? What do you think is going on?

Well, of course your `Sort()` function isn't working! It can't be much else, can it? Students! Now, take a deep breath and tell me what it is supposed to do. Sort, I know, but how?

Right, so you get this list of values from each object based on the field names. I like that, a bit of reflection. Nice. What happens with these values? What's this outer loop doing? I must say, this sorted flag is rather naff. I haven't seen anything that bad since your last C assignment. Come on, let's have it out.

OK, that's a loop till you've sorted it out. Ha ha, excuse the pun. So, getting serious, we have a loop on `i` for the objects being sorted and a loop on `j` for the fields in the objects. You appear to be moving through the fields and swapping objects if they are in the wrong order based on the fields.

Hello?

Yes, not got it yet? What causes the problem? It's this Jez Higgins fellow, is it not? Why him? What problem is he exposing? Take a look at that print out again – what's happening every time round the `while` loop? Yes, Jez and Kevlin are trading place. Why?

OK, which comes first, Jez or Kevlin? Right, Jez. Now, which comes first Higgins or Henney? Right, Henney. So?

Yes, you are not giving precedence to any of these fields: when they are in contradictory orders, they simply argue till the cows come home.

Woh, stop, halt! Not so fast. Let's look at what you could have done better, huh? Let's face it, there's some good stuff here, but also some code which would make the coolest cucumber sweaty.

Do you like writing sort algorithms? Not me – it's too easy to make a mess of it, as you did. I'd have used something else to sort it for me. Perhaps popped them into a `SortedDictionary`, or used `Array.Sort`.

I also think you could have used a better data set. Paul Simon and Paul Smith, with John Smith lurking there as well. Then you would have had to think about how to handle the 'nested' sort.

Hang on, what's the rush? Don't you think that `GetKeyForItem()` is a bit fishy sending back an object array? Wouldn't that be `GetKeysForItem()`? How can an object array be a key?

Names, names, these youngsters don't know what they can mean...

Commentary

It doesn't seem from the response to this critique that there are a lot of C# programmers in ACCU – or perhaps there are, but they are either busy or shy.

The most 'interesting' piece of this code was the sort algorithm. I literally laughed out loud the first time I saw this code (maybe I should get out more…) As Simon points out, it is almost always better to use a standard library for sorting than writing the code yourself. The original problem presented by the writer is that adding another name to the list stops the sorting working.

The first problem is that the inmost loop is supposed to the comparing the keys, and swapping over the `i` and `i+1` items. This the logic has one check – checking if `key1[j]` is greater than `key2[j]` for each element of the keys.

Let's see how this works in the test program. If the first key is `("Henney", "Kevlin")` and the second is `("Higgins", "Jez")` then the first comparison is `false` and the second is `true` (`"Kevlin" > "Jez"`). The code swaps them over, and then starts again, this time checking `("Higgins", "Jez")` against `("Henney", Kevlin")`. This time the first check is `true`, so the names are swapped – again!

The missing piece of the algorithm is that only if `key[j]` is equal to `key2[j]` should the next element of the key be checked. Fixing the code is easy, simply add an `else` clause as shown:

```
IComparable key =
  key1[j] as IComparable;
if (key != null) {
  if (key.CompareTo(key2[j]) > 0) {
    base.RemoveAt(i);
    base.Insert(i-1, object2);
    sorted = false;
    break;
  }
  else if (key.CompareTo(key[2]) < 0) {
    break;
  }
```

However the code is so inefficient and broken that I'm not sure I want to fix it. Let's unpack the algorithm. (1) The code loops until the collection is sorted. (2) On each loop every pair of items is checked until the first pair that is out of order. (3) This pair of items is swapped and the loop restarts. This is a very poor sorting algorithm already but in addition the code to fetch the key (`GetKeyForItem`) uses runtime reflection to get the key on every invocation. Leaving aside whether this is a good idea or not, each key is being fetched many times in the loop with an associated overhead.

The class as written is also broken: after inserting the first `TestColl` into the collection the dictionary-like indexing method won't find it again:

```
object[] key = new object[]{"Orr", "Roger"};
object found = coll[ key ];
```

This code throws a `KeyNotFoundException` which might surprise you at first. The reason though is that using an `object[]` as a key is not really viable – the `Equals` method on arrays just checks if the object references are identical. We need to be able to compare keys based on the values.

So the first change is to create a `Key` class that holds the `object[]` and provides a sensible `Equals` method, looking something like this:

```
public override bool Equals( Object obj ) {
  Key key2 = obj as Key;
  if ( key2 == null ) {
    return false;
  for ( int i = 0; i != key.Length; ++i ) {
    if ( ! key[i].Equals( key2.key[i] ) ) {
      return false;
    }
  }
  return true;
}
```

Then the `FieldCollection` can be changed to take a `Key` rather than an `object[]`.

Now we can think about a better way to do the sorting! The first refactoring is to move the comparison of keys into the `Key` class itself:

```
public int CompareTo( Object obj ) {
  Key key2 = obj as Key;
  int result = 0;
  for ( int i = 0; i != key.Length; ++i )
  {
    Object key = (IComparable)key[i];
    result = key.CompareTo(key2.key[i]);
    if (result != 0)
      break;
  }
  return result;
}
```

This makes the sort code simpler, but let's think about a better way.

The C# runtime includes a set of useful `Sort` methods in the `System.Array` class. The one that fits best here is:

```
public static void Sort (
  Array keys,
  Array items )
```

So the sort method becomes: create an array of the keys, create an array for the items, sort the keys and items and then re-fill the collection with the sorted values. Here is an example:

```
public void Sort() {
   Object[] items = new Object[Count];
   Dictionary.Values.CopyTo(items, 0);

   Object[] keys = new Object[Count];
   for ( int i = 0; i != Count; ++i ) {
     keys[i] = GetKeyForItem(items [i] );
   }

   Array.Sort(keys, items);

   Clear();
   foreach ( object item in items ) {
     Add( item );
   }
}
```

The resultant code is a lot simpler to understand, and also is a lot faster to execute although it does require a little bit more memory during the sort. For interest I sorted a collection of 1,000 names: it took less than 0.4s with the new algorithm and over 3 hours with the old, fixed, one. I could probably sort 1,000 names faster without a computer!

The Winner of CC 50

I liked the entry Simon supplied, which covered a lot of the problems with the code and the programmer's attitude while also being entertaining. There being no other entrant I duly declare Simon the winner.

Code Critique 50

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

For a slight change I presented a C# critique last time in the hope that a different language might encourage some new readers to attempt their first entry, but as this failed I'll revert the usual C/C++ !

Listing 2

```
#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"));
}
```

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?

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. ■

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.

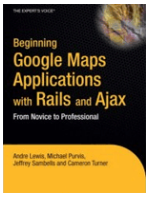
After you've made your choice, email me and if the book checks out on my database, you can have it. I will instruct you from there. Remember though, if the book review is such a stinker as to be awarded the most un-glamourous "not recommended" rating, you are entitled to another book completely free.

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

Beginning Google Maps Applications with Rails and Ajax

Andre Lewis, Michael Purvis, Jeffrey Sambells, Cameron Turner, Apress, ISBN 1590597877

Reviewed by Simon Sebright



It took me a while to review this book, as its topic is quite specific and I didn't know enough of the technologies to make sense of it. So, in the meantime, I got a couple of books on Ruby and Rails, and then continued the review.

Google have published the API for their web-based maps technology, allowing you create your own web sites/applications. This book takes you through the process of creating the 'Hello World' Google maps application, adding interaction via Ajax, and goes on to cover some advanced mapping concepts, which you may need to produce a heavily-used site, or to provide your own datasources. Geocoding is covered, using a number of sources including Google's own and Yahoo's.

Source code is provided online, although it didn't always match what was in the book, and it wasn't clear which stages of the chapter the various folders referred too, but it did build and run.

The positioning is that you know Rails and therefore Ruby and are comfortable creating and running Rails applications. They do give you some guidance, and I managed to muddle through. JavaScript and Ajax are introduced on a more gentle basis, as is the Google maps API, which is documented in more detail as an Appendix.

Generally, I found the book competent, and described the concepts well.

The different chapters sometimes build on the previous ones, sometime start something new, so you have to pay attention to what the code is and where it is supposed to go.

Using some of the ideas, I had a play with a couple of ideas I had and made some good progress. A lot of bits and pieces are required for these applications, particularly if you want them to look good. There's Rails, with Ruby, Rake, etc. Then the maps API is JavaScript-based.

Then there's CSS to present your pages, then there's more JavaScript to do some presentation where CSS isn't rich enough, some Ajax to talk to the server, etc. For that reason, I sometimes found myself a bit lost as to where to turn to change something. As mentioned above, the code online didn't always tally with what was in the book, and where all these bits and pieces come together, that did make a difference. Perhaps that illustrates a weakness of this type of application.

Overall, though, if you are planning to produce a website based on mapping concepts, this would be an excellent book to start with. For others, as it was for me, it's an interesting topic to read for its own sake, but doesn't have enough depth of any particular technology (ruby, javascript, Ajax, etc.) to serve as a reference. That said, if you have some knowledge of these things, having them all put together to create a working application might be something useful to read about here.

Moving to Free Software

by Marcel Gagne, published by Addison Wesley, ISBN 0-321-42343-7

Reviewed by Ian Bruntlett

I've been putting free software onto a mental health charity's (http://www.contactmorpeth.org.uk/) clients' PCs for nearly two years now and shipped about a hundred free PCs to them. I've personally built up a small library of useful programs and so I looked forward to reviewing this book and discovering new F/OSS gems.

This book comes with a DVD of F/OSS programs and, in general, it dedicates a chapter to each program. Some exceptions are 1) all the games are bundled together into one chapter and 2) OpenOffice isn't completely covered (but it does give chapters to Writer, Calc, Impress and Base)

To cut a long story short, here are the subjects and relevant programs that come with the book.

- Internet: FireFox (web browser), Thunderbird (email client), Gaim (IM), Skype (VOIP), NVU (Web Site Design)
- OpenOffice.org: Writer (word processor), Calc (spreadsheet), Impress(similar to MS PowerPoint), and Base (similar to MS Access)
- Audio: CDex (CD Ripper and Audio Converter), Audacity (Podcasts), Juice (Podcasts)
- Graphics: GIMP (like MS Paint), Inkscape (vector graphics), Scribus (DTP)
- Utilities : 7-Zip, (compressing files), SpyBot (anti-spyware), ClamWin (antivirus)
- Linux : Ubuntu Linux. One of the easier Linux distros available.

To finish it off, the following games are provided:

PlanetPenguin Racer, FreedroidRPG, Armagetron Advanced , Super Tux, BZFlag, Fish Fillets : Next Generation, Neverball and Neverputt, SolarWolf and Flightgear

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

- **Computer Manuals** (0121 706 6000) www.computer-manuals.co.uk
- **Holborn Books Ltd** (020 7831 0022) www.holbornbooks.co.uk
- **Blackwell's Bookshop**, Oxford (01865 792792) blackwells.extra@blackwell.co.uk

So that's what the book is good at. What are its weaknesses?

This book should mention:

- which versions of Windows the programs are compatible with
- what are the minimum hardware requirements
- what other programs do they rely on (GIMP relies on GTK)

The author really should have some way of enabling readers/users of this book to contact the author to tell him about incompatibilities and share workarounds – possibly a web site with a forum on. Don't expect to give this book to a non-technical friend and have all the programs work. You'll need to do some technical handholding and an internet connection is definitely needed to download apps that aren't fully on the disk (the DVD has GIMP on it but not the GTK toolkit that GIMP requires).

Verdict: Recommended.

Fundamentals of Global Positioning System Receivers: A Software Approach

by James Bao-Yen Tsui, Published by John Wiley & Sons, ISBN 0471381543

Reviewed by Colin Paul Gloster

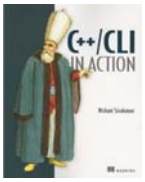
This is a fairly small and accessible book on the techniques necessary for making your own civilian GPS receiver. MATLAB is the language used and though the principles are transferable, the presented source code would not be efficient enough to use in an embedded product. The book's code could probably be used for verifying your own implementation, though I have not checked whether the book's code really works. The emphasis in the book is on explanations which are easy to understand in preference to optimal algorithms.

Overall, the book's aims seem to have been achieved, though most of the equations contain one- or two-letter variable names (many of which are from ancient Greek) and the naming policy for the MATLAB code is not much better. For example, one variable is called `rao` which is described in a comment as 'the pseudo-range' but the Greek letter rho is used for this in the body of the book. Perhaps rao is a synonym for rho but I have not found it in other books, and anyway, `pseudorange` would had been a better variable name. Another variable is named `erro` and I do not know what benefit the author perceived he was gaining by not typing error instead. On page 30 a clue is given as to why inappropriate names were used... an upside-down question mark appears where a 'less than' sign was supposed to be (as this is the absurd default treatment by LaTeX of the < character).

Section 2.14 is dominated by a discussion of how to obtain optimum precision from an unpopular less precise technique and does not give the impression that it is computationally easier. As mentioned above, I have not scrutinized the programs. Nor have I scrutinized the calculations. I did notice that it was incorrectly claimed on page 51 that 2 divided by 0.683 is approximately 29.28 whereas 2 divided by 0.0684 equals approximately 29.23 is closer to the truth. Somehow, the order of magnitude difference between 0.683 and 0.0684 did not propagate through to the final answer. A similar intermediary mistake leaving the final answer unaffected is on page 34. Table 3.1 contains mistakes but the associated text and numbers in the main body of the chapter are fine. On page 34, something which could pedantically be classified as a mistake is a measurement of a solid angle (a three dimensional counterpart to an angle) in degrees instead of steradians (as a steradian is a unit of solid angle instead of angle, it is different from a radian). It is fairly clear though: it is like saying that the distance between two towns is one hour.

The bulk of this review comprises of criticisms partially because I was showing off. The book is intended to be a practical introduction to the principles underlying GPS receivers and this is accomplished. So the practical necessity of converting GPS coordinates to layman's geographic coordinates is mentioned but a map of commercial value is something which you would need to acquire or make yourself. Other things are treated adequately but briefly, for example fewer than three pages are given over to distortions caused by the ionosphere. The ionospheric part of the book is just for the typical exploitation of GPS as a positioning system instead of its alternative use as an instrument to measure slant total electron contents through the ionosphere.

C++/CLI in Action



By Nishant Sivakumar, published by Manning, ISBN 1932394818

Reviewed by Seb Rose

Expert Visual C++/CLI: .NET for Visual C++ Programmers

By Marcus Heege, published APress, ISBN 1590597567

Reviewed by Seb Rose

Both books are targeted at experienced Visual C++ developers and they both have extensive coverage of getting native and managed code to interoperate. This is no surprise, since it's unlikely that many people will use C++/CLI to write new code that targets .NET, due to the added complexity and (relatively) poor support from the Visual Studio IDE.

I read Heege's book first and found it clear, concise and very dry. There aren't many jokes in the 352 pages, but I don't think I found a redundant paragraph either.

The first chapter gives an overview of the .NET landscape and how C++/CLI fits into it. He gets right into details of source file and object file compatibility and gives you a flavour of the power and flexibility of this binding.

The next 5 chapters cover all aspects of the managed environment. It's not rocket science, but is necessary to give an old-school C++ programmer enough knowledge to see what's going on. At the end of these chapters you will be able to write managed C++/CLI programs.

Chapter 7 shows how to extend native Visual C++ projects with managed code. There's in depth coverage of compilation models, compiler switches and exception handling. There's also a handy step-by-step guide to modifying your Visual Studio project to get it all working, which includes trouble shooting tips that really help. (Try finding Configuration Properties|Linker |Advanced|CLR Thread Attribute on your own when the CLR initializes the wrong COM apartment!)

Chapter 8 shows you how to create hybrid types that (appear to) have native and managed members. It's all smoke and mirrors (until/unless the unification hinted at in Herb Sutter's Design Rationale gets implemented), but is fairly simple to grasp. There are even some lovely MFC macros that allow your native types to handle managed events.

Chapter 9 dives deep into the mechanics of interoperation, covering thunks, double thunking and calling conventions (among others). There are quite a lot of pretty pictures of managed-unmanaged transitions and another hefty dose of Common Intermediate Language (CIL, latterly known as MSIL). He walks through each transition type and, though you'll need to go back to it several times before it sinks in, he makes it clear and comprehensible.

Chapter 10 covers wrapping native libraries so that they can be accessed by managed callers. He discusses design concerns that aren't strictly anything to do with C++/CLI or even .NET, and touches on CLS compliance. There's clearly going to be a lot of marshalling going on as objects make their way across the managed/unmanaged boundaries and he covers the library facilities that make this easier. This has all got a lot neater in Visual Studio 2008 with the template driven `marshall_as` library.

Chapter 11 introduces some techniques and library facilities for managing resources including IDisposable, Finalizers and the hugely useful SafeHandle. He also covers how to handle the various .NET asynchronous exceptions in an approved way, which is probably more than most developers need to know.

By the time you get to chapter 12 (the last chapter) you'd hope things were getting so esoteric that you could just skim over it, but you'd be wrong. Here he covers application startup and describes the various idiosyncracies of initializing the CRT depending on compilation model and whether you've built a DLL or an EXE.

The first appendix is a really useful utility that helps you modify machine settings so that you can run mixed-mode applications from network shares. The second appendix details a small app he wrote to measure the performance of thunks.

Finally, the index is adequate, and APress offer a PDF version of the book for US$10.

Having read Heege's book a couple of times I dived in Sivakumar's. The style couldn't be more different. It's conversational and repetitive. Each section ends by summarising what the next section is going to tell you. The use of similes is (to my mind) excessive. I don't want to be told that "Doing X is like putting a Chrysler Fender on a Ford Escort", but that's exactly the sort of thing I was told – over and over again.

The first chapter starts by setting the C++/CLI scene but doesn't really get into object/source file compatibility and jumps into the syntax on page 13, with the ubiquitous 'Hello, world'. The rest of chapter 1, and chapters 2 and 3 stick with the C++/CLI extensions.

Chapter 4 tackles mixed mode programming and mixed types. It also covers part of the marshalling library that Heege omitted – marshalling between function pointer and delegates.

Chapter 5 continues covering marshalling and talks about thunking, but not in any great detail. It then goes on to cover wrapping a native library and accessing COM objects. It's all quite brief, but you do feel that the ground has been covered.

Chapter 6 details mixing Windows Forms with MFC.

Chapter 7 explains how to target WPF with C++/CLI. It then goes on to show how you can host WPF in a native C++ app and use a native control in a WPF app.

Chapter 8 gives a brief tour of using the Windows Communication Foundation (WCF) from C++/CLI. It goes into some detail about how to migrate a DCOM app to WCF.

The appendix gives a concise introduction to the .NET framework.

As you'll have gathered by now, I found Heege's book to be well worth reading and a valuable resource. Sivakumar's book, by contrast, is a looser, lighter book, that seems more interested in the latest 'cool' technologies, rather than a deep understanding of what is going on, but it does cover some material that isn't in Heege's book.

Foundations of Security

by Daswani, Kern, Kesavan; published by Apress, ISBN 1-59059-784-2 pp290

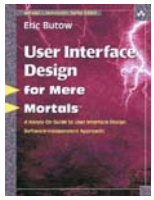
Reviewed by Mark Easterbrook

Conclusion: Highly Recommended.

Now that almost every device for which developers are generating code is connected to a network, and in most cases directly or indirectly to the Internet, it is essential for programmers to understand software security and how to protect against attack. Yet hardly a day goes by without a security incident of some kind, indicating that there is still a severe lack of security understanding in the software world. This book goes a long way to addressing this shortfall and should be essential reading for every software developer. Part one covers design principles: setting out the goals and how to design towards them. It also covers the well-intentioned but flawed approaches to security that lead to a false sense of security. Part two explores all the major forms of attack and describes how to counter them including many examples of secure, and not so secure, code. Part 3 is an Introduction to Cryptography and covers the subject in enough detail for the diligent designer to choose the correct encryption method. Finally part 4 contains appendices and references. The book is well written and provides a broad subject matter while still containing enough detail to go from beginner to skilled practitioner.

User Interface Design for Mere Mortals

by Eric Butow, published by Addison Wesley 2007, ISBN 0-321-44773-5



I personally would not recommend this book to IT professionals and probably not even to university students in IT.

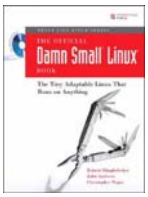
Let's start from the cover of the book. The front cover of the book states that it presents a Software Independent Approach to user interface design. On the back cover, the book is categorized as a User Interface Design/Software Design/Programming book. The book does not contain a single line of code and it is arguable if there is any software design in it at all.

Despite stating that the book is a software independent introduction to user interface design, web-based technologies are covered to some minimal detail where as desktop GUI development technologies like Visual Basic, Visual C++, C# and Delphi are totally ignored. The web technologies enumerated are only vaguely and implicitly categorized into front-end (browser-based) and back-end (server-side) technologies. Interestingly, the book categorises Java only as a web development technology regardless of the fact that Java is actively being used in visual desktop applications ranging from enterprise solutions to advanced military simulations

User interface prototyping has only been described with paper-based prototypes. Currently a number of GUI RAD tools are available that can help develop prototypes that provide a realistic look and feel. Executable GUI prototypes can be used in effective usability studies very early in the system development process thereby driving down the risk. In the simplest case, the use of interactive presentation tools for GUI prototyping provides better look and feel and also some initial usability testing than in case of prototyping with paper-based prototypes. Unfortunately, GUI RAD tools or techniques to develop interactive GUI prototypes have not been discussed at all.

The author does discuss some interesting topics like design patterns, principles and software postures. His categorization of websites is informative. There are also some useful tips on developing a business plan to justify usability testing and GUI development. However, this book does not contain sufficient information to allow readers to comprehensively understand characteristics of user interfaces and be able to design effective and user friendly interfaces. Unfortunately it turns out to be a complex mix of trivial and at times unrelated, inaccurate or incomplete information.

The Official Damn Small Linux Book, The Tiny Adaptable Linux That Runs on Anything



by Robert Shingledecker, John Andrews, and Christopher Negus, published by Prentice Hall, ISBN 0132338696

Reviewed by Giuseppe Vacanti

Damn Small Linux (DSL) is one of the 'tiny Linux' offerings available today. It was developed as a live CD system, but it has since been ported to boot from USB and compact flash, run inside a virtual machine, and also install itself on the hard drive. DSL packs a complete desktop system in 50MB, quite an achievement in comparison with what other more mainstream distributions can do (I have several times tried to do a minimal install of some more mainstream distributions and never been able to do it with less than 800MB).

This book is an extensive guide to DSL: one of the authors is the creator of DSL, another is the creator of DSL's extension system.

The book is divided in five parts. Part 1 deals with booting the live CD, configuring the system, and installing extensions. Here the authors explain in detail how to configure the system, and what applications are available. The description can be easily followed with very limited Linux knowledge.

Part 2 addresses way to run DSL other than as a live CD. From a pen drive, on a hard disk, or embedded in a virtual machine running on Windows: these and other possibilities are described.

Extending DSL by creating new packages, and making your own customized live CD are the topics coverd in Part 3. Here the learning curve gets steep for the Linux novice with details of building and installing software packages, editing the required configuration files,

View From The Chair

Jez Higgins
chair@accu.org

If I could have stalled this View for another week[1], I expect I could be writing about the invigorating effects of this year's conference. Right now, I just have anticipation and preparation. There are things to sort out for AGM, people I'd like to meet, conversations to be had. I got caught on the hop a little last year, and I'd rather avoid that if I can.

As an organisation, I believe ACCU has a great deal to offer people. We have, though, not been quite as good at letting people know as we might have. That is changing, thanks in large part to David's efforts as publicity officer, and to the work the groups in London, Cambridge, and elsewhere are doing. Since the conference sequesters so many of us in a hotel for four days, it's the obvious time to chew over what's gone on and to try a plan ahead. It's the spring! [2] It's natural to look to the future in spring!

Although, by the time you read this, the AGM will have passed and the committee and offices elected or re-elected, if there's something you feel ACCU could or should be doing do speak up. Want to publicise ACCU at your place of work or university? Would like to be the new Pete Goodliffe and write a column for CVu? Toying with setting up your own local group? Know of a project or organisation ACCU should be supporting? Suggesting something new can often seem daunting, but adapting to change is a fundamental part of software. If I as the Chair, or the committee as a whole, can make it happen, then we will try to make happen.

[1] Although if I had, Tim would probably have reached half way around the world to throttle me.

[2] In my garden, at least.

Membership Report

Mick Brooks
accumembership@accu.org

I've just finished preparing the reports which will be used to produce a new, and long-overdue, edition of the membership handbook. By the time you're reading this, I hope it will be ready to go live on the website. I think it's important that members are able to contact each other, and the handbook is one way to support that (the mailing lists, conference, and the new Facebook and LinkedIn groups are others). I want to thank you all for responding to my letters and emails asking you to update your contact preferences in readiness for this change.

A small number of members were concerned that, by taking inaction as permission to publish your data in a new form, we weren't treating your contact information with proper respect. We take our reponsibilities toward your data very seriously. I've tried to contact every member of the ACCU to ask them to update their preferences. Where I've been unable to do so, either because a member has withheld permission for us to email them, or because an email has bounced, I've removed permission for their details to appear in the handbook.

A significant proportion of you have chosen to opt-out of the new format handbook. This is understandable, and I hope that some of you will feel differently once you've seen exactly how the information will be made available. One advantage of the new system is that it can be updated more frequently: if you do change your mind, you won't have to wait a whole year before the change comes into effect.

If you want to comment on this (I'm particularly interested to hear why those who chose to opt-out did so), or need help with any aspect of your membership, please email me at accumembership@accu.org.

Advertising Officer Report

Seb Rose
ads@accu.org

Over the past few months, Tim Pushman of Gnomedia and myself have been enabling the ACCU website to accept paying advertisers. As you may have noticed the system went live a few weeks ago, with in house adverts as well as a trial run for Perforce. We could really do with more advertisers, so please consider recommending the site to your employers and/or business partners. ■

Book Reviews (continued)

compiling a new kernel, and burning a new bootable CD image.

In Part 4 we learn about complete DSL installations for a specific goal (the authors call these installations DSL projects): a music server, a VOIP station, and an Apache-MySQL-PHP server. This part will appeal to those willing to quite literally hack a system together; in fact the project descriptions start from the selection of suitable old hardware, to the installation of DSL, and the tweaking of various scripts.

The final part contains the appendices, one of which is the list of all packages available in DSL version 3.3. The book comes with a CD and its contents are described in the second appendix. The CD can be run as a live system; it also contains several other DSL boot images (all of those described in the book), two versions of the Windows-embedded version of DSL, various scripts and the additional software and scripts needed to work on the projects described in part 4.

The book is a comprehensive guide to DSL, addressing both the novice and the more advanced users.

Books of this type can become rapidly out of date: the book covers version 3.3, whereas the current DSL version is 4.2.4. The main concepts described in the book are likely to apply to the more recent DSL versions, although the details are most likely to differ.