The magazine of the ACCU

{cvu}

Volume 25 • Issue 2 • May 2013 • £3

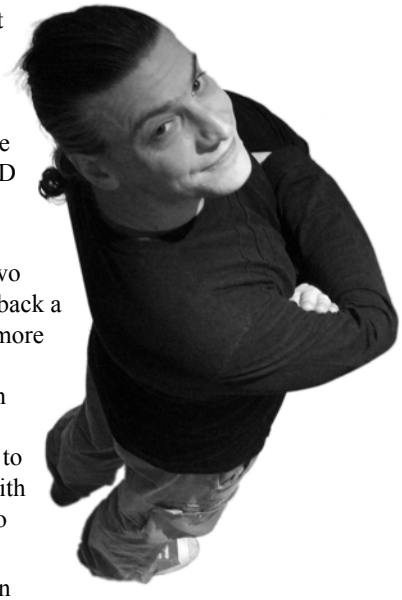
Features

Regulars

{cvu}
Volume 25 Issue 2
May 2013
ISSN 1354-3164
www.accu.org

The New Informs The Old

A few weeks ago, I finally finished reading the Freeman and Price book *Growing object oriented software, guided by tests*. I bought it a couple of years ago, and (for shame!) have only just got round to reading it. For even more shame, I decided once I'd finished it to read a book that's been left practically untouched for far longer, Kent Beck's *Test-driven development*. The reason I'd not got round to reading that one is that there's so much commentary and discussion about TDD – within ACCU, and on countless forums – I felt I'd already read it, in a way.



What was most interesting for me was the reading of these two books *back to back*, and in reverse order, so to speak. Going back a decade or so with Kent's book gave me new insights on the more modern practice, which also fed new comprehension of *Growing*..... The basic premise seems to be very similar, with the keeping of a to-do list, writing code test first, getting fast feedback, keeping the code 'clean'. The New has differences to the Old, for example, getting a 'walking skeleton' in place, with acceptance tests, which drives a slightly different approach to development.

Indeed, this difference of approach seems to have spawned an entire debate: that of 'Classic' (or 'Detroit') versus 'London-style' TDD. This seems to me to be similar to the differences between 'Mockists' and 'Classicists' as described by Martin Fowler in his article 'Mocks aren't stubs', but goes further than that. The idea of aiming first for a full end-to-end test (with the help of Mock Objects) drives design differently to beginning with the simplest piece of functionality that represents measurable progress, as in 'classic' TDD. I've heard this described as 'outside-in' design versus 'inside-out'.

I'm pretty sure I don't yet understand *either* approach well enough to comment on the better-ness of either one. However, one of those insights I mentioned that came from reading both books, was that the more modern approach looks to me like a natural progression of the classic approach, and that the use of Mock Objects to explore the relationships between collaborating objects is *complementary* to testing publicly visible state. Of one thing I am certain: whether you write tests in 'classic' or 'London' style is less important than your tests being clear and useful – and that you've written some!

STEVE LOVE
FEATURES EDITOR

The official magazine of ACCU

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

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

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

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

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

DIALOGUE

REGULARS

FEATURES

SUBMISSION DATES

C Vu 25.3: 1st June 2013
C Vu 25.4: 1st August 2013

Overload 116: 1st July 2013
Overload 117: 1st September 2013

WRITE FOR C VU

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

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

ADVERTISE WITH US

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

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

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

COPYRIGHTS AND TRADE MARKS

Bug Hunting

Pete Goodliffe implores us to debug effectively.

If debugging is the process of removing software bugs, then programming must be the process of putting them in.
~ Edsger Dijkstra

It's open season. A year-round season. There are no permits required, no restrictions levied. Grab yourself a shotgun and head out into the open software fields to root out those pesky varmints, the elusive bugs, and squash them, dead.

Well, it's not really as saccharin that. But sometimes you end up working on code in which you swear the bugs are multiplying and ganging up on you. A shotgun is the only response.

The story is an old one, and it goes like this: Programmers write code. Programmers aren't perfect. The programmer's code isn't perfect. It therefore doesn't work perfectly first time. So we have bugs.

If we bred better programmers we'd clearly breed better bugs.

Some bugs are simple mistakes that are obvious to spot and easy to fix. When we encounter these, we are lucky.

The majority of bugs, the ones we invest hours of effort tracking down, losing our follicles and/or hair pigment in the search, are the nasty, subtle issues. These are the odd surprising interactions, or unexpected consequences of the actions we instigate. The seemingly non-deterministic behaviour of software that looks so very simple. It can only have been infected by gremlins.

no matter how sound your code-writing regimen, some of those pernicious bugs will always manage to squeeze through the defences

This isn't a problem limited to newbie programmers who don't know any better. Experts are just as prone. The pioneers of our craft suffered; the eminent computer scientist Maurice Wilkes wrote in [1]:

I well remember [...] on one of my journeys between the EDSAC room and the punching equipment that 'hesitating at the angles of stairs' the realisation came over me with full force that a good part of the remainder of my life was going to be spent in finding errors in my own programs.

So face it. You'll be doing a lot of debugging. You'd better get used to it. And you better get good at it. (At least you can console yourself that you'll have plenty of chance to practice.)

An economic concern

How much time do you think is spent debugging? Add up the effort of all of the programmers in every country around the world. Go on, guess.

Greg Law (who provided me with the initial impetus to write this – as well as collating an amount of excellent material that I have wilfully stolen) points out that a staggering $312bn per year is spent on the wage bills for programmers debugging their software. To put that in perspective, that's two times all Euro-zone bailouts since 2008! This huge, but realistic, figure comes from research carried out by Cambridge University's Judge Business School [2].

You have a responsibility to fix bugs faster: *to save the global economy*. The state of the world is in your hands.

It's not just the wage bill, though. Consider all the other implications of buggy software: shipping delays, cancelled projects, the reputation damage from unreliable software, and the cost of bugs fixed in shipping software.

An ounce of prevention

It would be remiss of any article on debugging to not stress how much better it is to actively prevent bugs manifesting in the first place, rather than attempt a post-bug cure. *An ounce of prevention is worth a pound of cure.* If the cost of debugging is astronomical, we should primarily aim to mitigate this by not creating bugs in the first place.

This, in a classic editorial sleight-of-hand, is material for a different article, and so we won't investigate the theme exhaustively here.

Suffice to say, we should always employ sound engineering techniques that minimise the likelihood of unpleasant surprises. Thoughtful design, code review, pair programming, and a considered test strategy (including TDD practices and fully automated unit test suites) are all of the utmost importance. Techniques like assertions, defensive programming and code coverage tools will all help minimise the likelihood of errors sneaking past.

We all know these mantras. Don't we? But how diligent are we in employing such tactics?

> Avoid injecting bugs into your code by employing sound engineering practices. Don't expect quickly-hacked out code to be of high quality.

The best bug-avoidance advice is to not write incredibly 'clever' (which often equates to complex) code. Brian Kernighan states:

Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.

Martin Fowler reminds us:

Any fool can write code that a computer can understand. Good programmers write code that humans can understand.

Bug hunting

Beware of bugs in the above code;
I have only proved it correct, not tried it.
~ Donald Knuth

Being realistic, no matter how sound your code-writing regimen, some of those pernicious bugs will always manage to squeeze through the defences and require you to don the coder's hunting cap and an anti-bug shotgun. How should we go about finding and eliminating them? This can be a Herculean task, akin to finding a needle in a haystack. Or, more accurately, a needle in a needle stack.

Finding and fixing a bug is like solving a logic puzzle. Generally the problem isn't too hard when approached methodically; the majority of bugs are easily found and fixed in minutes. There are two 'vectors' that make a bug hard to fix: how reproducible it is, and how long it is between the cause of the bug itself (the 'software fault') and you noticing. When a

PETE GOODLIFFE

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

bug scores high on both, it's almost impossible to track down without sharp tools and a keen intellect.

If you plot frequency versus time-to-fix you get a curve asymptotically approaching infinite time to fix. In other words, the hard bugs are few in number, but that's where we will spend most of our time.

There are a number of practical techniques and strategies we can employ to solve the puzzle and locate the fault.

The first, and most important thing, is to investigate and characterise the bug. Give yourself the best raw material to work with:

- Reduce it to the simplest set of reproduction steps possible. Sift out all the extraneous fluff that isn't contributing to the problem, and only serves to distract.
- Ensure that you are focusing on a single problem. It can be very easy to get into a tangle when you don't realise you're conflating two separate – but related – faults into one.
- Determine how repeatable the problem is. How frequently do your repro steps demonstrate the problem? Is it reliant on a simple series of actions? Does it depend on software configuration, or the type of machine you're running on? Do peripheral devices attached make any difference? These are all crucial data points in the investigation work that is to come.

In reality, when you've constructed a single set of reproduction steps, you really have won most of the battle.

Here are some useful debugging strategies:

Lay traps

You have errant behaviour. You know a point when the system seems correct (maybe it's at start-up, but hopefully a lot later through the repro steps), and you can get it to a point where its state is invalid. Find places in the code path *between* these two points, and set traps to catch the fault.

Add assertions or tests to verify the system invariants that must hold. Add diagnostic print-outs to see the state of the code so you can work out what's going on.

As you do this, you'll gain a greater understanding of the code, reasoning more about the structure of the code, and will likely add many more assertions to the mix to prove your assumptions hold. Some of these will be genuine assertions about invariant conditions in the code, others will be assertions relevant to this particular run. Both are valid tools to help you pinpoint the bug. Eventually a trap will snap, and you'll have the bug cornered.

> Assertions and logging (even the humble printf) are potent debugging tools. Use them often.

Many of these diagnostic logs and assertions may be valid to leave in the code after you've found and fixed the problem.

Learn to binary chop

Aim for a *binary-chop* strategy, to focus in on bugs as quickly as possible. Rather than single-stepping through code paths, work out the start of a chain of events, and the end. Then partition the problem space into two, and work out if the middle point is good or bad. Based on this information, you've narrowed the problem space to something half the size. Repeat this a few times, and you'll soon have homed-in on the problem.

Employ this technique with trap-laying. Or with the other techniques below.

Employ software archaeology

Software archaeology describes the art of mining through the historical records in your version control system. This can provide an excellent route into the problem; it's often a simple way to hunt a bug.

Determine a point in the near past of the codebase when this bug didn't exist. Armed with your reproducible test case, step forwards in time to determine which code changeset caused the breakage. Again, a binary chop strategy is the best bet here.

Once you find the breaking code change, the cause of the fault is usually obvious, and the fix self-evident. (This is another compelling reason to make series of small, frequent, atomic check-ins, rather than massive commits covering a range of things at once.)

Do not despise tests

Invest time as you develop your software to write a suite of unit tests. This will not only help shape how you develop and verify the code you've initially written. It acts as a great early warning device for changes you make later; it acts like the miner's canary – the test fails long before the problem becomes complex to find and expensive to fix.

These tests can also act as great points from which to begin debugging sessions. A simple, reproducible unit test case is a far simpler scaffold to debug than a fully running program that has to spin up and have a series of manual actions run to reproduce the fault. For this reason, it's advisable to write a unit test to demonstrate a bug, rather than start to hunt it from a running 'full system'.

Once you have a suite of tests, consider employing a *code coverage* tool to inspect how much of your code is actually covered by the tests. You may be surprised. A simple rule of thumb is: if your test suite does not exercise it, then you can't believe it works. Even if it looks like it's OK now, without a test harness then it'll be very likely to get broken later.

> Untested code is a breeding ground for bugs. Tests are your bleach.

When you finally determine the cause of a bug, consider writing a simple test that clearly illustrates the problem, and add it to the test suite *before* you really fix the code. This takes some genuine discipline, as once you find the code culprit, you'll naturally want to fix it ASAP and publish the fix. Instead, first write a test harness to demonstrate the problem, and use this harness to prove that you've fixed it. The test will serve to prevent the bug coming back in the future.

Invest in sharp tools

The are many tools that are worth getting accustomed to, including memory checkers like electric fence, and swiss-army knife tools like Valgrind. These are worth learning *now* rather than reaching for them at the last minute. If you know how to use a tool before you have a problem that demands it, you'll be far more effective.

Learning a range of tools will prevent you from cracking a nut with a pneumatic drill.

Of course, the tool of debugging champions is the *debugger*. This is the king of tools that allows you to break into the execution of a running program, step forwards by a single instruction, or step in – and out of – functions. Some advanced debuggers even allow you to step backwards. (Now, that's real voodoo.)

In some circles there is a real disdain for the debugger. *Real programmers don't need a debugger.* To some extent this is true; being overly reliant on such a tool is a bad thing. Single-stepping through code mindlessly can trick you into focusing on the micro, rather than thinking about the overall shape of the code.

But it's not a sign of weakness. Sometimes it's just far easier and quicker to pull out the big guns. Don't be afraid to use the right tool for the job.

> Learn how to use your debugger well. Then use it at the right times.

Remove code to exclude it from cause analysis

When you can reproduce a fault, consider removing everything that doesn't appear to contribute to the problem to help focus in on the offending lines of code. Disable other threads that *shouldn't* be involved.

Remove subsections of code that do not look like they're related. It's common to discover objects indirectly attached to the 'problem area', for example via a message bus or a notifier-listener mechanism. Physically disconnect this coupling (even if you're *convinced* it's benign). If you still reproduce the fault, you have proven your hunch about isolation, and have reduced the problem space.

Then consider removing or skipping over sections of code leading up to the error (as much as makes practical sense). Delete, or comment out blocks that don't appear to be involved.

Cleanliness prevents infection

Don't allow bugs to stay in your software for longer than necessary. Don't let them linger. Don't dismiss problems as *known issues*. This is a dangerous practice. It can lead to *broken window syndrome* [3]; making it gradually feel the norm and acceptable to have buggy behaviour. This lingering bad behaviour can mask the causes of other bugs you're hunting.

Stop for a moment and consider if there are other related problems lurking in that section of code

One project I worked on was demoralisingly bad in this respect. When given a bug report to fix, before managing to reproduce the initial bug you'd encounter ten different issues on the way that all also needed to be fixed, and may (or may not) have contributed to the bug on question.

Oblique strategies

Sometimes you can bash your head against a gnarly problem for hours and get nowhere. It's important to learn when you should simply stop and walk away. A break can give you fresh perspective.

This can help you to think more carefully. Rather than running headlong back into the code, take a break to consider the problem description and code structure. Go for a walk and step away from the keyboard. (How many times have you had those 'eureka' moments in the shower? Or in the toilet?! It happens to me all the time.)

Describe the problem to someone else. Often when describing any problem (including a bug hunt) to another person, you instantly explain it to yourself and solve it. Failing another actual, live, person, you can follow the *rubber duck strategy* described by the Pragmatic Programmers [4]. Talk to an inanimate object on your desk to explain the problem to yourself. It's only a problem if the rubber duck starts to talk back.

Don't rush away

Once you find and fix a bug, don't rush mindlessly on. Stop for a moment and consider if there are other related problems lurking in that section of code. Perhaps the problem you've fixed is a pattern that repeats in other sections of the code. Is there further work that you could do to shore up the system with the knowledge you just gained?

Non-reproducible bugs

Having attempted to form a set of reproduction steps, sometimes you discover that you can't. It's just not possible. From time to time we uncover nasty, intermittent bugs. The ones that seem to be caused by *cosmic rays* rather than any direct user interaction. These are the gnarly bugs that take ages to track down, often because we never get a chance to see them on a development machine, or when running in a debugger.

How do we go about finding these?

- Keep records of the factors that contribute to the fault. Over time you many spot a pattern that will help you identify the common causes.
- As you get more information start to draw conclusions. Perhaps identify more data points to keep in the record.
- Consider adding more logging and assertions in beta/release builds to help gather information from the field.
- If it's a really pressing problem, set up a test farm to run long running-soak tests. If you can automate driving the system in a representative manner then you can accelerate the hunting season.

There are a few things that are known to contribute to such unreliable bugs. You may find they provide hints as to where to start investigating:

- Threaded code; as threads entwine and interact in non-deterministic and hard-to-reproduce ways, they often contribute to freaky intermittent failures. Often this behaviour is very different when you pause the code in a debugger, so is hard to observe forensically.
- Network interaction, which is by definition laggy and may drop or stall at any point in time. Code that presumes access to local storage works (because, most often, it does) will not scale to storage over a network.
- The variable speed of storage (spinny disks, database operations, or network transactions) may change the behaviour of your program, especially if you are balanced precariously on the edge of timeout thresholds.
- Memory corruption, where your aberrant code overwrites the stack or heap, can lead to a myriad of unreproducible strangenesses that are very hard to detect. Software archaeology is often the easiest route to diagnose these errors.

Conclusion

Debugging isn't easy. But it's our own fault. We wrote the bugs.

Effective debugging is an essential skill for any programmer. ■

Acknowledgments

The inspiration for this article came from a conversation I had with Greg Law about his excellent ACCU 2013 conference presentation on debugging. Greg's company, Undo Software, creates a most impressive 'backwards debugger' that you may want to look at. Check it out at undo-software.com.

References

[1] Maurice Wilkes, *Memoirs of a Computer Pioneer*. The MIT Press. 1985. ISBN 0-262-23122-0

[2] Cambridge Research puts the global cost of debugging at $312billion annually. reference. http://undo-software.com/content/press-release-7

[3] Broken Windows Theory http://en.wikipedia.org/wiki/Broken_windows_theory

[4] Andrew Hunt and David Thomas, *The Pragmatic Programmer*. Addison Wesley. ISBN 0-201-61622-X.

Questions

1. Assess how much of your time you think you spend debugging. Consider every activity that isn't writing a fresh line of code in a system.
2. Do you spend more time debugging new lines of code you have written, or on adjustments to existing code?
3. Does the existence of a suite of unit tests for existent code change the amount of time you spend debugging, or the way you debug?
4. Is it realistic to aim for bug-free software? Is this achievable? When is it appropriate to genuinely aim for bug-free software? What determines the optimal amount of 'bugginess' in a product?

Tar-Based Back-ups

Filip van Laenen rolls his own with some simple tools.

A few months ago, I found out that I had to change the back-up strategy on my personal laptop. Until then I had used Areca [1], which in itself worked fine, but I was looking for something that could be scripted and used from the command line, in addition to be easier to install and maintain. As often is the case in the Linux world, it turned out that you can easily script a solution together on your own using some basic building blocks. For this particular task, the building blocks are Bash [2], tar, rm and split, together with sha256sum and cmp to build a conditional copying function.

Why use a script?

What was my problem with Areca? First of all, from time to time, Areca had to be updated. In the Linux world, this is usually a good thing, but not if the new version is incompatible with the old archives. This can also cause problems when restoring archives, e.g. from one computer to another, or after a complete reinstallation of the operating system. Furthermore, since Areca uses a graphical user interface, scripting and running the back-up process from the command line (or crontab) wasn't possible.

Notice that these problems were generic, and not particular to Areca. Before deciding to script a solution together, I looked for an alternative solution that was scriptable and easy to install, but without success. That is, except for the suggestions to build my own solution using tar.

Getting started

Listing 1 shows the start of my tar-based back-up script. It starts with a shebang interpreter directive to the Bash shell. Then it checks the number of arguments that were provided to the script – it should be exactly one, otherwise the script exits here. Next it sets up four environment variables: a base directory in **BASEDIR**, the back-up directory where all archives will be stored in **BACKUPDIR**, the number of the current month (two digits) in **MONTH**, and the first argument passed to the script in **LEVEL**. The **LEVEL** variable represents the back-up level, i.e. 1 if only the most important directories should be archived, 2 if some less important directories should be archived too, etc…

Backing up a Directory

Next we define a two parameter function that backs up a particular directory to a file. Listing 2 shows how this function looks, together with some examples of how it can be used. First it logs to the console that it's going to back up. Next it uses tar to do the actual archiving. Notice that the output of tar is redirected to a log file. That way we keep the console output tidy, and at the same time can browse through the log file if something went wrong. That's also why we included **v** (verbosely list files processed) in the option list for tar, together with **c** (create a new archive), **p** (preserve file permissions), **z** (zip) and **f** (use archive file). Finally the function creates a SHA-256 [3] digest from the result. This digest can be used to decide whether two archive files are identical or not without having to compare large, multi-GB files.

The variable **MONTH** is used to create rolling archives. In Listing 2, the directories `bin` and `dev` will always be backed up to the same archive file,

Listing 1

```
#!/bin/bash
## Creates a local back-up. The resulting files
# can be dumped to a media device.

if [[ $# -ne 1 ]]; then
   echo "Usage:"
   echo "  `basename $0` <LEVEL>"
   echo "where LEVEL is the back-up level."
   exit
fi

BASEDIR=/home/filip
BACKUPDIR=${BASEDIR}/backup

MONTH=`date +%m`

LEVEL=$1
```

but for the `Documents` and `Thunderbird` directory, a new one will be created every month. Of course, if the script is run a second time during the same month, the archive file for the `Documents` and `Thunderbird` directory will be overwritten. Also, the same will happen when the script is run a year later: the one year old archive file will then be overwritten with a fresh back-up. If you want some other behaviour, like e.g. a new archive every week or every day, you simply have to define your own variable and use date to set it. Tailor to your needs in your own back-up script!

Listing 3 shows how **LEVEL** can be used to differentiate between important and often-changing directories on the one hand, and more stable directories you do not want to archive every time you run your script on the other hand. Currently my back-up script has three levels, but I'm considering splitting off the small archives from level 1 in a separate level, so I could add a line to crontab to take a quick back-up of some important directories once every day.

Splitting large files

Next, I'd like to split large files into chunks that are easier to handle when transferring them to external media. This makes it easier to move archives between computers or to external media. Listing 4 shows a function that splits a large file into pieces of 4 GB (hence the magic number 4,294,967,296 = 4 × 230), together with a loop that finds all files that should be split.

Let's start with a look at the function that splits the files. It receives one parameter, the path to the file. The first thing the function does is to extract

Listing 2

```
function back_up_to_file {
  echo "Backing up $1 to $2."
  tar -cvpzf ${BACKUPDIR}/$2.tar.gz\
      ${BASEDIR}/$1 &> ${BACKUPDIR}/$2.log
  sha256sum -b ${BACKUPDIR}/$2.tar.gz\
      > ${BACKUPDIR}/$2.sha256
}

back_up_to_file bin bin
back_up_to_file dev dev
back_up_to_file Documents Documents-${MONTH}
back_up_to_file .thunderbird/12345678.default\
                  Thunderbird-${MONTH}
```

FILIP VAN LAENEN

Filip van Laenen is a chief technologist at the Norwegian software company Computas. He has a special interest in software engineering, security, Java and Ruby, and likes to do some hacking on his Ubuntu laptop in his spare time. He can be contacted at f.a.vanlaenen@ieee.org

Listing 3

```
# Backup of directories subject to changes
if [ ${LEVEL} -ge 1 ]; then
   back_up_to_file bin bin-${MONTH}
   back_up_to_file Documents Documents-${MONTH}
   back_up_to_file .thunderbird/12345678.default\
                    Thunderbird-${MONTH}
   back_up_to_file dev dev-${MONTH}
   …
fi

# Backup of relatively stable directories
if [ ${LEVEL} -ge 2 ]; then
   back_up_to_file Drawings Drawings
   back_up_to_file Photos/2010 Photos-2010
   back_up_to_file Movies/2013 Movies-2010
   back_up_to_file .fonts fonts
   …
fi

# Backup of stable directories
if [ ${LEVEL} -ge 3 ]; then
   back_up_to_file Music Music
   …
fi
```

the file name from the path, so that we can log to the console in a nice way which file we're going to split. Next it removes any chunks it finds from the previous run, using option **f** to suppress any error messages in case there aren't any chunks present. Then it does the splitting into chunks of 4 GB, using option **d** to create numeric suffixes instead of alphabetic. This means that if the function would split a file called `dev.tar.gz`, the names of the resulting chunks would be `dev.tar.gz.00`, `dev.tar.gz.01`, etc… Finally, when the function is done, it removes the original file, because we don't need to have it around any more.

The function is called inside a loop, which goes through all files having the `.tar.gz` extension. For each file it uses stat to calculate the total size (**-c%s**), and then compares it to 4 GB. If the file is larger, our function to split the file is called.

Done

Finally, at the end of the script, we write to the console that we're done (**echo "Done."**). I like to do that to indicate explicitly that everything went well, especially since this script can take a while.

Storing the back-ups

There's a little detail in Listing 1 that we haven't dealt with yet: where do we store the back-ups? The script as it stands can be used to create local back-ups, i.e. putting the back-up files on the same disk as the original data, on the one hand, or write the back-ups directly to an external disk on the other hand. Since I have enough space on my hard disk, I like to create the back-up files locally first, and then plug in the external disk to transfer the files. That's also why I create SHA-256 digests, so I can detect when a back-up file hasn't changed and doesn't need to be transferred to the external disk.

Listing 4

```
function split_large_file {
   FILENAME=$(basename $1)
   echo "Going to split ${FILENAME}."
   rm -f $1.0*
   split -d -b 4294967296 $1 $1.
   rm $1
}

for f in ${BACKUPDIR}/*.tar.gz
do
   FILESIZE=$(stat -c%s $f)
   if (( $FILESIZE > 4294967296 )); then
      split_large_file $f
   fi
done
```

Listing 5

```
function copy_files {
   rm -f "${TARGETDIR}/$1.tar.gz"*
   for f in ${BACKUPDIR}/$1.tar.gz*
   do
      FILENAME=$(basename $f)
      echo "Copying ${FILENAME}."
      cp $f "${TARGETDIR}"
   done
   cp ${BACKUPDIR}/$1.log "${TARGETDIR}"
   cp ${BACKUPDIR}/$1.sha256 "${TARGETDIR}"
}

for f in ${BACKUPDIR}/*.sha256
do
   FILENAME=$(basename $f)
   BASEFILENAME=\
      `echo ${FILENAME} | sed -e 's/.sha256$//'`
   cmp -s ${BACKUPDIR}/${FILENAME}\
      "${TARGETDIR}/${FILENAME}" > /dev/null
   if [ $? -eq 0 ]; then
      echo "Skipping ${BASEFILENAME}."
   else
      copy_files ${BASEFILENAME}
   fi
done
```

Listing 5 shows how the back-up files we just created can be copied conditionally to an external drive. It loops through all the SHA-256 digests in the directory with the back-up files, and compares them to the SHA-256 digests in the target directory using **cmp** (silently, though the option **s**). If both files exist, and their content is the same, **cmp** will return 0. In that case, we don't need to copy files to the target directory, and can continue with the next SHA-256 digest. Otherwise we call the function that copies the set of files associated to the SHA-256 digest.

The function to do that takes one parameter: the basename of the SHA-256 digest file, but without the extension. The set of files we then want to copy consists of either the back-up file as a whole, or the different chunks resulting from the split function, in addition to the log file and of course the SHA-256 digest. We therefore have to start by removing the old back-up file or the chunks from the split function. Next, we copy the back-up file or the chunks in a small loop that lets us log to the console what we're doing. Finally, we also copy the log file and the file with the SHA-256 digest. Notice that copying the SHA-256 digest file is the last thing we do: if the script is interrupted, we want to be sure that the next run will try and copy this file set again.

The code in Listing 5 forms the body of its own script, separate from the code in the other listings. In fact, the script contains only three more things: the definition of **BACKUPDIR** and **TARGETDIR**, and writing to the console that we're done. It assumes that we want to keep a copy of the back-up files on our hard disk, hence the use of **cp** to transfer the files to the target directory. If you'd rather move the back-up files to the target directory, you should not only use **mv** instead of **cp** to transfer the files, but also remember to remove the set of files from the back-up directory in case of identical SHA-256 digests. ■

References

[1] See http://www.areca-backup.org/

[2] See http://www.gnu.org/software/bash/

[3] See http://en.wikipedia.org/wiki/SHA-2

ACCU Conference 2013

Chris Oldwood shares his experiences from this year's conference.

It's April once again and that can only mean one thing – apart from the school holidays and Easter eggs – it's the ACCU Conference. This year saw one of the biggest changes to the conference – a new venue. And not just a new hotel but in a new city too! For the last 5 years I've only ever been to the same hotel in Oxford and so with much trepidation I headed down to Bristol. One of my biggest 'worries' was what was going to replace all those long standing traditions, like 'Chutneys Tuesday'? But hey, we're all agile these days so we should embrace change, right?

Wednesday

Once again I didn't get to partake in one of the tutorial days on the Tuesday which was a shame as they looked excellent as usual. Instead I made my way down on the Wednesday and arrived in the early afternoon. That meant I missed lunch, but also more importantly the keynote from Eben Upton about the Raspberry Pi and a talk from Jonathan Wakely about SFINAE. The comments coming through on Twitter about these, and the other parallel talks, generated much gnashing of teeth as I cursed my late arrival.

Not wanting to take things lightly I dived head-first into Johan Herland's session about Git. I've done a little messing around with Git and have read the older 1st edition O'Reilly book but I wasn't sure whether I'd really understood it. Luckily Johan walked us slowly through how Git works in theory, and then in practice. I'm glad I saw this as it seems I was on the right track but he explained some things much better than the book. I also got to quiz him in the bar later about how some Subversion concepts might translate to Git, or not as it seems, which was priceless.

Next up, was Pete Goodliffe doing the live version of his *C Vu* column on Becoming a Better Programmer. This was split into two parts with the first being Pete discussing what we even mean by 'better'. He provided some of his thoughts and there were the usual array of highly entertaining slides to back them up – you are always guaranteed a good show. The second part was provided by various speakers (chosen by Pete) who got to spend 5 or so minutes discussing a topic that they believe makes them a better programmer. Unsurprisingly these varied greatly from the practical, such as Automation (Steve Love), to the more philosophical – The Music of Programming (Didier Verna). The audience got to have a quick vote on what they felt was the most useful and Seb Rose's Deliberate Practice got the nod.

Once the main sessions have finished for the day the floor is opened up to everyone in the guise of Lightning Talks. These are short 5 minute affairs where anyone can let off steam, share a tip or plug something (non-commercial). Even though it was only the first evening there was a full program with talks about such topics as Design Sins (Pete Goodliffe), C++ Active Objects (Calum Grant), BDD with Boost Test (Guy Bolton King), Communities (Didier Verna) and an attempt at Just a Minute from Burkhard Kloss. With 12 talks in total it was a good start.

Although not directly part of the ACCU conference, the Bristol & Bath Scrum Group held an evening event afterwards where James Grenning talked about TDD. Although it had been a long day already I couldn't resist squeezing one more talk in, especially from someone like this. Being aimed at a wider audience than just developers meant there were an interesting assortment of questions afterwards which was useful. One in particular was the common question of writing the test first versus immediately after which always causes a interesting debate.

Thursday

A full English breakfast and plenty of coffee set me up for the day and I was greeted with my first 2013 keynote, courtesy of Brian Marick. He started with an interesting tangent about crickets and how they tune in to a mate and eventually got onto the topic of how to cope with the inevitable natural decay older programmers will suffer from. The thing that has stuck with me most is the advice of converting 'goal attainment' to 'maintaining invariants'. This he explained by showing how a baseball fielder might try to catch a ball by moving himself so he sees a linear trajectory rather than trying to anticipate a parabolic path. This was one of the most enjoyable keynotes I've seen.

I didn't have much choice in what I went to after Brian as it was my turn to step up to the plate. This was my third year of speaking and I'd like to say I might finally be getting the hang of it. At least, I don't think anyone fell asleep.

Unbeknownst to me until I checked my Twitter feed afterwards but there was a small bug in the code on one of my slides. This made my next choice easy – The Art of Reviewing Code with Arjan van Leeuwen. There was plenty of sound advice here, particularly around the area of getting started in code reviews where it's important to make both parties comfortable to avoid a sense of personal attack. As Arjan pointed out, time is often the perceived barrier to doing reviews, but it's reminded me how valuable it can be.

That was only a short session and the other 45 minutes I spent with Ewan Milne as he discussed Agile Contracts. Although I don't get involved (yet?) in that side of the process I still find it useful to comprehend the other parts of an agile approach. Understanding how the different forms of contract attempt to transfer the risk from one side to the other was enlightening – especially when you consider the role lawyers try to play in the process. Ewan normally has his hands full with organising the lightning talks so it was good to see him speak for longer than 30 seconds.

My final session for the day was to

CHRIS OLDWOOD

Chris started as a bedroom coder in the 80s, writing assember on 8-bit micros. Now it's C++ and C# on WIndows. He is the commentator for the Godmanchester Gala Day Duck Race and can be reached at gort@cix.co.uk or @chrisoldwood

be spent with Michel Grootjans. With a title of Ruby and Rails for n00bs I felt that suited my knowledge of Ruby right down to the ground and hoped I would get to see what some of the fuss is about. It actually turned out to be way more useful than I expected because Michel developed a simple web app using a full-on TDD approach too. Not only did I get a small taste for what Ruby and Rails is about but I also saw someone develop a different sort of application using different tools in a more enterprise-y way.

Once more, after the main sessions had completed, most of us convened to the main hall to listen to another round of lightning talks. This time there was a total of 13 topics with an even wider range than the day before. Notably for me, given my attendance at an earlier session on Git, was a rant from Charles Bailey about Git being evil. There were also complaints about poor variable naming (Simon Sebright) and why anyone would use C++ when D exists (Russel Winder, naturally). On the more useful front we saw Dmitry Kandalov implement an Eclipse plug-in in 5 minutes and a C++ technique that seems close to C#'s async/await mechanism (Stig Sandnes). The abusive C++ award though goes to Phil Nash with his `<-` operator for implementing extension methods. Oh, and Anders Schau Knatten used 'Science' to help us decide that C# is in fact the best programming language.

Friday

What better start to the day than a keynote from the very person we have to thank for C++ – Bjarne Stroustrup. It's been some years since he graced the ACCU conference with his presence and so like many I was looking forward to what he had to say about the modern state of C++. His presentation was generally about the new features we now have in C++ 11 as he had a separate session planned for C++ 14. However there was as much about how the established practices (e.g. RAII) are still the dominant force and critical to its effectiveness. Naturally there were plenty of questions and he pulled no punches when airing his opinion on the relationship between C and C++.

With my C++ side ignited I felt it was only right that I attend Nico Josuttis' talk about move semantics and how that plays with the exception safety guarantee of a function like `push_back()`. He entered the murkier depths of C++ to show how complex this issue is for those who produce C++ libraries. When someone like Nico says C++ is getting 'a little scary' you know you need to pay attention. My 'moment of the conference' happened here when, in response to a question for Nico about the `std::pair` class, Jonathan Wakely instantly rattled off the C++ standard section number to help him find the right page…

We all love writing fresh, new code, but many of us spend our lives wallowing in the source code left to us by others. Cleaning Code by Mike Long was a session that showed you why refactoring is important and what some of the tools and techniques you can use to help in the fight against entropy. This was a very well attended talk and rightly so with a good mix of the theoretical and practical. One tool in particular for finding duplicate code certainly looked sexy and will definitely be getting a spin.

After another round of coffee I decided to close the day off by listening to the *C Vu* editor (Steve Love) explain why C# is such a Doddle to learn and use. Yes, his tongue very firmly placed in his cheek. As someone who uses C# for a living it's easy to forget certain things that you take for granted with something like C++, such as the complexity guarantees of the core containers. Generics also came in for a bit of a bashing as a watered down version of templates. Anyone who thinks the world of C# is dragon free would have done well to attend.

The final set of lightning talks took their cue from the volcano fiasco a few years ago. Back then, due to speaker problems caused by a lack of air transport, a set of 15 minute lightning keynotes were put together instead and that's the length these ones adopted. Seb Rose opened the proceedings with a response to an earlier lightning talk about whether the term 'passionate' is a useful one for describing the kind of people we want to work with, given its dictionary definition. Much nodding of heads suggested he was probably right. He was followed by me trying to show how many of the old texts, such as the papers by David Parnas, are still largely relevant today. And, more importantly they're often cheap. Tom Gilb was next up to answer a question I had posed in my talk about quantifying robustness. Let's face it, we knew he would. Finally Didier Verna got to extend his earlier slot on The Pete Goodliffe Show to go into more detail about the similarities he sees between music and programming. I've never really given Jazz a second thought before, and even though we only got a 30 second burst of his own composition my interest is definitely piqued.

The Friday evening always plays host to The Conference Dinner, which is a sort of banquet where we get to spend a little more time mingling with the various speakers and attendees. This is a perfect opportunity to corner a speaker and ask some questions you didn't get a chance to earlier. Jon made sure the tables regularly got mixed up to keep the flow of people moving between courses which helps you mix with people you might not normally know. After the dinner there was the Bloomberg Lounge to keep us entertained through the night, if you fancied staying up until silly o'clock.

Saturday

There was another change to the session structure this year as the Saturday keynote was moved to the end of the day; instead the normal sessions started earlier. Sadly I overdid the conference dinner again and so an early start was never really on the cards.

How to Program Your Way Out of a Paper Bag seemed like the ideal eventual start to the day. Frances Buontempo had sold the idea well – is it possible to actually write a program to get out of a paper bag? Obviously there was a certain amount of artistic licence, but ultimately she did it, and along the way we got to find out a whole lot about machine learning. I was a little worried there might be a bit too much maths at that time of day but it was well within even my meagre reach.

My final session of the conference was to be with Hubert Matthews – A History of a Cache. This was a case study of some work he been involved in. The session had a wonderful narrative as he started by explaining how the system was originally designed, and then went on to drop the bomb on how he needed to find a huge performance boost with the usual array of 'impossible' constraints. Each suggested improvement brought about a small win, but not enough by itself and that's what made it entertaining. It also goes to show what can be achieved sometimes without going through a rewrite.

Epilogue

I keep expecting the magic of this conference to wear off, but so far it seems to be holding fast. I have looked around at some of the other conferences but I'm just not as impressed by the content or the price for that matter. I thought the new venue worked well and even though it was a little further to travel it wasn't exactly onerous. More of the talks were filmed this year and so hopefully I should be able to catch up on some of those I missed. With 5 concurrent sessions running in each time slot you're never going to get to see everything you want to, but that's just another reason to keep coming back year-after-year – to try and catch up on everything you've missed in previous years. Of course in the meantime the world has moved on and there's another load of new stuff to see and learn! ■

Writing a Cross Platform Mobile App in C#

Paul F. Johnson uses Mono to attain portability.

A brief piece of history

Many years back, Ximian (a small bunch of very nice people) decided to write an open source version of the .NET language based on the ECMA documentation. Initially for Linux, it soon spread to Mac, BSD and many other platforms (including Windows). This was good and fine. Novell then bought Ximian and signed what was considered (in the non-SuSE part of the open source community at least) as a deal with the devil – the devil being Microsoft.

Time moved on. Novell was bought out and so Xamarin was formed, their task, to carry on developing the open source Mono framework which was fast growing to be a recognised force for good.

While all of this was going on, Google moved into the mobile phone business with Android and Apple released their iPhone. Android (as you may know) has a Linux kernel at its heart and apps are coded in Java. iPhones use Objective-C for the language of choice. Google controlled its app store and Apple, in true Apple fashion, pretty much dictated under the guise of 'quality' what could and could not be distributed through them.

This is fine and dandy with one problem – as with the old 8-bit systems of old, if you wanted your app to run on both iPhone and Android, you had to do a lot of work to port the code over, that or employ that rare breed developers that can work in both Objective C and Java.

It doesn't take a genius to realise that if a company can come up with a method to write once, deploy many as was the case with .NET, then they would win the day and praises be sung. Step forth Xamarin. Using the mono framework, they released .NET for both iPhone and Android. While the UI aspect is not the same, a large amount of core functionality could be moved between the platforms with minimal work (it is after all just using the .NET framework that we all love and use) reducing both development time and final cost. For the iPhone, as the code generated reverts back to ObjC and is linked against Apple's SDK, apps created with Monotouch (the iOS version) are available in the iOS store.

What this small series is going to show is how simple it is to achieve both an iPhone and Android version of the same app with essentially the same code. I will be porting some code I wrote [1] quite a few years back to run on both platforms. It isn't going to do anything amazing, but will allow you to download, read and reply to your gmail.

Xamarin have released versions of monotouch and monodroid that will run on the emulator (Android) or simulator (iOS) [2] so you can see and test the final product. The source code for these articles is held online [3].

My recommendation is that you install Xamarin Studio to code with. While there are plugins for VisualStudio 2010 and 2012, my experience with them has not been great, whereas Xamarin Studio is rock solid.

Let's get on with it then

The basis of this app is communicating with the Google servers to allow a user to read and reply to their emails. To do this, we need a basic SMTP and POP3 system. SMTP is supported natively, POP3 isn't, but it's not difficult to code a small POP3 library that allows access to the facilities.

PAUL F. JOHNSON

Paul used to teach and was one time editor of a little known magazine called *C Vu*. He now writes code professionally for a living – primarily for Android and iOS, but only ever in .NET thanks to Xamarin.Android and Xamarin.iOS.

A word of warning

When writing code that will work between both iOS and Android, it is not only the UI that needs to be considered. Monotouch for .NET developers is a much simpler system to use. Instantating new classes which generate new views is very similar to how it is done in a standard Winforms application

```
NewView nv = new NewView(params);
```

will create a new instance of `NewView` with whatever parameters are needed to be passed in – it is essentially the same as in a winforms application.

Android development is not like this. For Android, the safest way to think about how an app is structured is that there are a lot of small apps (called Activities) that you need to get to work together. While you can certainly pass certain objects between activities (the likes of `string`, `int`, `bool` etc), passing the likes of classes or bitmaps is not going to happen.

To start a new activity

```
Intent i = new Intent(this, typeof(class));
StartActivity(i);
```

where `class` is the name of the activity class being started.

Passing simple objects can be done with

```
string hello = "Hello";
…
i.PutExtra("name", hello);
```

and read back in the receiving class using

```
string message =
   base.Intent.GetStringExtra("content");
```

Alright, it's not rocket science, but it leads to two problems; portability (it's not available in iOS) and propagation (the next activity will also have to have the same `PutExtra`/`GetExtra` code to receive the data).

This difficulty can be overcome by using either a standard interface block or better than that, a public static class. The big advantage of having the static class is that generics, arrays, bitmaps and anything else that can be bundled into a static class can be used. It is also completely portable between the platforms – as long as nothing platform specific is included in there of course!

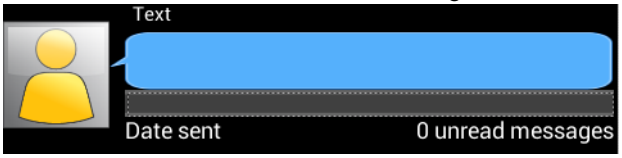
Of course, there is nothing to stop instantiation between classes on Android in the more usual .NET form, but it will not fire up the activity, so no view is shown unless a bit of extra legwork is done. I will be avoiding that route for this series!

UI design

This app will not win any design awards, but then it's not meant to. It is simple and functional. The UI is greatly different between iOS and Android. To that end, I will concentrate on that aspect for the remainder of this article.

Android

The way to think about how to design for Android is to think in either vertical or horizontal boxes. Take the following



While it would seem a simple enough design, it has to be considered along the lines of boxes within boxes *viz*

We have a horizontal outer, next layer are two horizontals. The one on the right now has 4 verticals with the bottom one having two horizontals in. Planning can be a bit tricky on deciding which way the layout has to be, but it's not that bad. By default, a new layout contains a vertical **LinearLayout**.

Within each layout, you can pretty much put any type of view (most of the widget classes are derived from the View class, so you have a **TextView**, **EditView**, **ImageView** and so on). Android here becomes very similar to .NET in that the views are similar to the standard .NET views (for example **TextView** = **Label**, **EditView** = **TextBox**, **ImageView** = **PictureBox**), but like the .NET widgets, these views can be 'themed' using XML.

A view can be used by any number of activities on Android. Monodroid (thankfully) comes with a UI designer as part of the Monodevelop or VS plug in.

```
SetContentLayout(Resource.Layout.foo);
```

And that's it to get the UI to display.

Attaching events to widgets is simple as well. My UI has a **TextView** called **textView**.

```
TextView text =
   FindViewById<TextView>(Resource.Id.textView);
```

To attach a click event can be done in a number of ways

```
1. text.Click += delegate {…;};
2. text.Click += (object s, EventArgs e) => {
     someMethod(s,e); }
3. text.Click +=  delegate(object s, EventArgs e)
   {…;};
4. text.Click += HandleClick;
```

Each has a different purpose

1. Used for performing a particular task where the event parameters can be completely ignored (for example, performing a calculation or calling another method with any number of parameters, the return value of which is used in some way)
2. Used as a both a standard event and also it allows for methods to be overloaded (so pass **s**, **e** and say an **int**, **string** and **bool** as well)
3. Similar to (1), except now the event parameters are being passed in and can therefore be accessed and worked with.
4. **HandleClick** does what it says on the tin. This is call to the method **HandleClick**. The object and eventargs are passed into the call. This is handled outside of the **OnCreate** method.

iOS

As you may expect from Apple, everything on their devices is a rich experience and for that to happen, the developer has to be free to allow their mind to roam, not be constrained by box limitations and generally whatever they want to go, can go.

All iOS UI development had to be done on a Mac. This may change in the future, but for now, it's safe to say that all design will be done using XCode. XCode is free to download from Apple. The most recent version of Xamarin.iOS will allow you to code for Apple devices on a PC, as long as there is a networked Mac for XCode to be accessed on.

With XCode, you can put things wherever you like and don't have the same rigid design constraints as you do for Android or Windows Phone.

Unlike Android though, the communication between the iOS UI and application is a bit more complex. With iOS, you have two types of interface, an *outlet* and an *action*. Don't let the names fool you; an outlet is the one that reacts when you click on it, the action is the receiver.

A widget can be both an action and outlet. Take the following code

```
btnClickMe.TouchDown += delegate {
   btnClickMe.SetTitle ("Clicked",
     UIControlState.Highlighted);
};
```

Here, **btnClickMe** is both the outlet (**TouchDown** event) and the action (**SetTitle**). When creating the UI though, it is usually sufficient to say if an object is an outlet or an action.

iOS calls the view into existence when the class it belongs to is called into existence. However, there are some considerations to add along to that.

```
public override void ViewDidLoad()
```

This method is called immediately after the class has been instantated. At this point, you can either add in what you want the outlets to do, or call another method to do that for you (which can be preferable sometimes). This is similar to the Android **OnCreate** method.

```
public override void ViewDidUnload()
```

called when the class is finished with. This removes the view, so freeing up the memory it previously occupied.

Unlike Android, the types of (say) Click are different. Typically in Android, you have Click. In iOS, there are 9 different Touch events covering cancel, drag, clicks inside of an object and even a plain normal click (**TouchDown**). It could be considered overkill, or it could be considered as giving the developer far greater control over every aspect of the development cycle. Either way, there are a lot of them.

Memory management

This is not an issue for iOS. The **ViewDidUnload()** method removes the view, frees the memory and makes life easy.

Not so on Android.

The reason for this is easy enough. Monodroid is C# on top of Java. Think of it more as a glue layer than anything. When an object is created in C#, the C# GC disposes of the object when it's done with. The problem is this. When dealing with the UI, the glue creates a Java object for (say) the **TextView** widget and everything to do with that widget is handled through the glue layer. At the end of it's life, the C# GC will clear away only the reference it has used. It does not dispose of the underpinning object from the Java layer – the Java object sits there, hogging memory until the app falls over dead.

For Android, there are two simple methods to ensure you don't run out of memory

1. Whenever you can, if a process is memory intensive (typically anything to do with graphics), employ something like **using (Bitmap bmp = CreateBitmapFromFile(filename)) { }**. Once out of scope, the memory is freed up.
2. At the end of the activity, explicitly dispose of the objects by calling the GC.

   ```
   protected override void OnDestroy()
   {
     base.OnDestroy();
     GC.Collect();
   }
   ```

   will do this for you.

That's enough for this time. Next time I'll start to look at code and how it differs between the platforms to do the same task. ■

References

[1] http://www.all-the-johnsons.co.uk/csharp/email.html
[2] http://xamarin.com/trial
[3] http://www.all-the-johnsons.co.uk/accu/mobile/article1.zip

Let's Talk About Trees

Richard Polton puts n-ary trees to use parsing XML.

This article will show how to define a tree data structure in both C# and F# and then will proceed to create a tree and load the contents of an XML file containing SPAN data into it. Let's start with a quick recap over tree structures.

The classic binary tree, which contains a value at each node, might be represented in C# as shown in Listing 1.

As can be seen, the data structure is defined recursively. That is, it is defined in terms of itself. Therefore, any node contains zero, one (because the code has allowed null in the setter) or two subtrees in addition to a value of type `T`. Such a tree might be initialised using

```
var t = BinaryTree.Node( 5,
  BinaryTree.Node( 8, BinaryTree.Leaf(9),
      BinaryTree.Leaf(7)),
     .Node( 2, BinaryTree.Leaf(1),
      BinaryTree.Leaf(3)));
```

In F# we might define the tree structure as

```
type tree =
  | Node of 'T * tree * tree
  | Leaf of 'T
```

which also makes it clearer that any single (sub-)tree is either a branch point, containing both a value and left and right branches, or a leaf, containing only a value. We might then create an object of this type using

Listing 1

```
public class BinaryTree<T>
{
  public T Value { get; private set; }
  public BinaryTree<T> Left { get; private set; }
  public BinaryTree<T> Right { get; private set; }

  public BinaryTree(T value, BinaryTree<T> left,
                    BinaryTree<T> right)
  {
    Left = left;
    Right = right;
    Value = value;
  }
}
public static class BinaryTree
{
  public static BinaryTree<T> Node<T>(T value,
    BinaryTree<T> left, BinaryTree<T> right)
  {
    return new BinaryTree<T>(value, left, right);
  }

  public static BinaryTree<T> Leaf<T>(T value)
  {
    return new BinaryTree<T>(value, null, null);
  }
}
```

RICHARD POLTON

Richard has enjoyed functional programming ever since discovering SICP and feels heartened that programming languages are evolving back to LISP. He likes 'making it better' and enjoys riding his bike when he can't. He can be contacted at richard.polton@shaftesbury.me

Listing 2

```
public class NaryTree<T>
{
  public Tuple<T,List<NaryTree<T>>> Node
    { get; private set; }
  public List<NaryTree<T>> SubTrees
    { get { return Node.Item2; } }

  public NaryTree(T value,
                  List<NaryTree<T>> subTrees)
  {
    Node = Tuple.Create(value, subTrees);
  }
}
```

```
let t = Node (5,
  Node ( 8, Leaf 9, Leaf 7),
  Node ( 2, Leaf 1, Leaf 3))
```

See [1] and [2] for further information on the definition and traversal of binary tree structures.

Let us now generalise this to an n-ary tree. In C# we might write this as Listing 2, which is roughly the C# equivalent of the F# tree definition given by the discriminated union (see [3] for a discussion of Algebraic Data Types)

```
type tree =
  | Node of 'T * tree list
```

Let us now pause awhile and divert our attention to the reason why this subject presented itself in the first place. XML.

XML – it's supposed to be the Holy Grail of data formats, easily consumed by both the computer and the lucky human reader. I recently had the distinct pleasure of working with some SPAN XML files [4] published by the Australian Stock Exchange. These files are freely available for download and a snippet from one of these files is reproduced here.

This snippet (Listing 3), lightly edited, was extracted from `ASXCLEndOfDayRiskParameterFile130305.spn`

As might be expected, the XML represents a hierarchical data set. The highest-level element in the snippet, **`clearingOrg`**, contains both simple

Listing 3

```
<clearingOrg>
  <ec>ASXCLF</ec>
  <name>ASX Clear Futures</name>
  <curConv>
    <fromCur>AUD</fromCur>
    <toCur>USD</toCur>
    <factor>0.000000</factor>
  </curConv>
  <pbRateDef>
    <r>1</r>
    <isCust>1</isCust>
    <acctType>H</acctType>
  </pbRateDef>
  <pbRateDef>
    <r>4</r>
    <isCust>1</isCust>
    <acctType>H</acctType>
  </pbRateDef>
</clearingOrg>
```

Listing 4

```
let rec readClearingOrg (reader:System.Xml.XmlReader) acc =
  match reader.Name with
  | "ec" -> SpanXMLClearingOrg.Ec
    (reader.ReadElementContentAsString()) :: acc
    |> readClearingOrg reader
  | "name" -> SpanXMLClearingOrg.Name
    (reader.ReadElementContentAsString()) :: acc
    |> readClearingOrg reader
  | "curConv" -> (SpanXMLClearingOrg.CurConv
    (readCurConv (reader.ReadStartElement() ;
     reader) [] )) :: acc
    |> readClearingOrg (reader.ReadEndElement() ; reader)
  | "pbRateDef" -> (SpanXMLClearingOrg.PbRateDef
    (readPbRateDef (reader.ReadStartElement() ; reader) [] )) :: acc
    |> readClearingOrg (reader.ReadEndElement() ; reader)
  | _ -> acc
```

and complex data elements, eg **name** and **pbRateDef** respectively. (Before you ask, no, I didn't change the names of the elements. They really are called **ec** and **r**!)

We want to load the XML and parse it into a data structure using F#. We want to do this so that we can subsequently query the data set automatically instead of having to rely on eyeballs and Notepad. I say Notepad because, although the data sets are not especially large, they do appear to be large enough to cause both Internet Exploder's and Visual Studio's XML renderers to fail, which leaves the ever-faithful Notepad as our key inspection vehicle.

The first attempt at parsing this XML made use of discriminated unions like the below:

```
type SpanXMLClearingOrg =
  | Ec of string
  | Name of string
  | CurConv of SpanXMLCurConv list
  | PbRateDef of SpanXMLPbRateDef list
```

given prior similar definitions for **SpanXMLCurConv** and **SpanXMLPbRateDef**. This layout maps trivially to the XML representation and so building a parser for this is very easy.

Whilst it may be possible to parse this XML using LINQ to XML using a dictionary as demonstrated in [5], in this version of the parser, the XML is read using recursive functions such as seen in Listing 4.

As can be seen, the function makes use of an accumulator (see article in previous *CVu* for a quick intro or htdp.org [6]) to store the state of the parsed structure up until the current point. In the example code the state is called **acc** and is a list of **SpanXMLClearingOrg**. Other than that the parser simply repeats the above form for each data structure that is to be read from the XML. That is, compare the name of the current element with one of a set of possible names and take the appropriate action, which is one of converting the element value to a specific data type, eg **int**, or reading an embedded data structure, eg **PbRateDef**. The result is then prepended to the accumulated list of data structures loaded thus far and then the function is called again. If the name of the current element does not match any of the possible names then the function exits returning the accumulated list to the caller. Thus the tree is built up as the XML is consumed.

In the end we had a tree of data but unfortunately it turned out to be very difficult to query. So much so, in fact, that an alternative representation was sought.

Instead of the 'natural' mapping from XML to structures as shown above, we chose to use a traditional functional tree data structure.

In the literature, for example [7], functional tree structures are presented for binary trees. They look like this:

```
type tree =
  | Leaf of string
  | Node of tree * tree
```

In other words, every node in the tree contains either two further trees or a value, in this case a string. Note that the data structure is defined recursively.

Our tree, however, is slightly different. It is not a binary tree but is an *n*-ary tree (where *n* depends on the actual location in the tree). Also each node has one or more values. Additionally, each of the different levels of the tree, at least in the XML, can only be created from a well-defined subset of data types. We can tackle the fact that a node has a value as well as a subtree by defining our tree structure as

```
type tree =
  | Leaf of string
  | Node of string * tree * tree
```

This is a bit unsatisfactory, though, primarily because of the unnecessary distinction between **Leaf** and **Node** as all the nodes in our tree contain data. However, we can modify the definition to accomodate this and extend to multiple sub-trees using

```
type tree<'T> =
  | Node of 'T * tree<'T> list
```

Et voilà! Well, almost. We now have a recursive tree structure whose every node can contain a datum as well as zero (because the list can be empty) or more sub-trees. The next challenge is how to render our data structures such that they will fit in this new tree.

We can solve this trivially by defining an algebraic data type to be the union of all the possible types of data that can be stored at a node. In order to retain the structure of the original XML, we choose to create records (which are like 'C' structures) that hold the data values and then the union refers to all the record types. So, for example, we can define the record

```
type SpanXMLCurConv =
{
  FromCur : string
  ToCur : string;
  Factor : float;
}
```

to represent the currency conversion data element **curConv**. This XML element does not contain any complex XML elements itself but its parent, the XML element **clearingOrg**, clearly does. We choose to represent **clearingOrg** as the record

```
type SpanXMLClearingOrg =
{
  Ec : string;
  Name : string;
}
```

Note that the nested complex XML elements are not stored within the record in this implementation (unlike in the first implementation of the parser). This is because we will be storing the nested complex elements in the list of sub-trees. However, we still need to define a union so that it is possible to store one of a number of distinct data types in the data value of the node. So we write

```
type nodeType =
  | ....
  | SpanXMLCurConv of SpanXMLCurConv
  | ...
  | SpanXMLClearingOrg of SpanXMLClearingOrg
  | ...
```

where the first of the two names in the union is the name of the discriminator and the second is the name of the type that is stored therein. Now we can rewrite our tree type definition as

```
type tree =
  | Node of nodeType * tree list
```

Listing 5

```
let findAllCurConv theTree =
  let rec findAllCurConv' theTree acc =
    match theTree with
    | Node (SpanXMLCurConv (_), _) as node -> node :: acc
    | Node (_, subTrees) -> subTrees |>
      List.collect
          (fun node -> findAllCurConv' node acc findAllCurConv' theTree []
```

Listing 6

```
let findCurConvFrom fromCur theTree =
  let rec findCurConvFrom' theTree acc =
    match theTree with
    | Node (SpanXMLCurConv (curConv), _) as node
      when curConv.FromCur = fromCur ->
        node :: acc
    | Node (_, subTrees) -> subTrees |>
      List.collect (fun node -> findCurConvFrom' node acc)
  findCurConvFrom' theTree []

let allConversionsFromGBP = findCurConvFrom "GBP" theTree
```

Listing 7

```
let findCurConvWithPath fromCur tree =
  let rec findCurConv tree acc path =
    match tree with
    | Node (SpanXMLCurConv (cc) as uNode, _) as node
      when curConv.FromCur = fromCur ->
        (node, (uNode :: path |> List.rev)) :: acc
    | Node (_, []) -> acc
    | Node (uNode, trees) ->
      trees |>
      List.collect (fun node -> findCurConv node acc
                                          (uNode :: path))
  findCurConv tree [] []
```

Listing 8

```
let findDivsWithPath pred tree =
  let rec findDivs tree acc path =
    match tree with
    | Node (SpanXMLDiv (div) as uNode, _) as node
      when pred div ->
        (node, (uNode :: path |> List.rev)) :: acc
    | Node (_, []) -> acc
    | Node (uNode, trees) ->
      trees |>
      List.collect (fun node -> findDivs node acc (uNode :: path))
  findDivs tree [] []

  let findDivs f tree = findDivsWithPath f tree |> List.map first
```

The advantage of a data structure of this form is the ease by which it can be traversed and, therefore, queried. Given the above definition, we can write queries to extract all **curConv** elements very simply (Listing 5).

If we want to find a specific conversion, say from GBP for example, then we could modify our function to take an extra parameter and to use this as a guard in the 'match' (Listing 6).

It couldn't be easier. This works because of the power of the F# pattern matching. This is analogous to the switch statement in C-style languages except that the pattern that is being matched is not constrained to compile-time constants. Type matching, as here, is commonplace, as are more sophisticated matches on the return values of functions. Look at **Functional.Switch** for an example of a similar construct in C# (both prior editions of *CVu* and functional-utils-csharp [8] on Google Code).

So it looks like the pain of transforming the XML into our new tree structure is going to pay dividends (boom! boom!). All that is missing now is that transformation. The 'read' functions all have the same format. On account of there being so many record types having such similar structure, we created a code generator to simplify the work. This code generator produces the basic reader function which we then manually modify to account for the nested structures. (This was a trade-off; time to code vs time to edit by hand, and the latter won the day.) For the terminally curious, the code generator lives in the span-for-margin project [9].

Notice that, although **findAllCurConv** is a very simple query function, it has a shortcoming in that it only returns those nodes which satisfy the criterion supplied and does not provide the route taken through the tree in order to reach them. We want to modify the function so that a path to each successful node is also returned.

First, then, we need to change the internal find function to return a 2-tuple, having the matching node and the path to the matching node as its components. This 2-tuple becomes our accumulator. Therefore, on a successful match we return

```
(node, (uNode :: path |> List.rev)) :: acc
```

where **node** is the Node which has been matched, **uNode** is the **SpanXML** record, **path** is a list of **SpanXML** records traversed to reach this point and **acc** is the accumulator. Note that we have to reverse the **path** list once we have a match because functional lists prepend new items to the head rather than append to the tail.

If the function fails to find a matching node, i.e. we have an unsuccessful termination condition, then at the bottom of a given branch we just return the current state of the accumulator.

In the 'inbetween' state where we have a node which does not match but is not a leaf node, i.e. it has a non-empty list of sub-trees, we need to prepend this node to the **path** and then call the recursive **find** function again for each of the subtrees under this node.

And so we can write Listing 7: **collect** is the F# analogue of **SelectMany**, or more precisely, **SelectMany** is based upon the algorithm encapsulated by **collect**. That is, given a function which accepts a single element and which returns a list, evaluate this function for every element in the container and flatten the results into a single list.

Now suppose we want to find the **Div** nodes in the tree which satisfy some predicate. We could write very similar code to **findCurConvWithPath**, changing only the **nodeType** name in the **match** (Listing 8) using, for example:

```
let divDateChk fromCur (curConv:SpanXMLCurConv) =
  curConv.FromCur = fromCur
findDivs (divDateChk "1-Apr-2013") theTree
```

Clearly, the **findNodeTypeWithPath** pattern will be repeated for all node types to be queried in the tree. Instead of copying the entire function perhaps there is some way we can generalise the **findN** function.

Active patterns [7] are the obvious choice here. This would leave us with

```
let findNodeWithPath actPattern f tree =
  let rec findNode tree acc path =
    match tree with
    | actPattern ....
```

but the problem with this is that it does not appear to be possible to pass an Active Pattern as a parameter to a function. If any of you know how to do this, my email address is in the byline. Otherwise, huh! So much for all functions being first-class objects in F#. Therefore, we would like to be

able to define a general Active Pattern which accepts a parameter. This parameter would then be the type name that we wish to check.

```
let (|Check|) theType input =
  ...
```

However, this quickly becomes unwieldy leading to a worse mess of code than we had in the original problem and so we must seek an alternative approach. Given that we are not going to be able to use an Active Pattern, let us pass instead a predicate-like function that returns an option (again, see previous *CVu* and Google Code [8]). Even though adopting this approach means that we will have to perform an additional pattern match step outside of our generic function, it should be an improvement. See Listing 9.

Listing 9

```
let findNodeWithPath pattern tree =
  let rec findNode tree acc path =
    match pattern tree with
    | Some(uNode,node) -> (node, (uNode :: path |> List.rev)) :: acc
    | _ ->
    match tree with
    | Node (_, []) -> acc
    | Node (uNode, trees) ->
      trees |>
        List.collect (fun node -> findNode node acc (uNode :: path))
  findNode tree [] []
```

In this function, **pattern** is a function with signature **(tree -> (nodeType * 'a) option)**. For example, the following function **divNode** could be used as the **pattern**.

```
let divNode input =
  match input with
  | Node (SpanXMLDiv (record) as uNode, _) as
    node -> Some(uNode,node)
  | _ -> None
```

However, this doesn't allow us to filter the **Div** nodes of interest as we can do so in **findDivsWithPath**. If we modify the function to accept an additional parameter then we can pass a curried function into the **find** function. So we write

```
let divNode f input =
  match input with
  | Node (SpanXMLDiv (record) as uNode, _)
    as node when f record -> Some(uNode,node)
  | _ -> None
```

where **divNode** has been redefined to accept a predicate **f**. Now we can write

```
let divs = findNodeWithPath (divNode fn) tree
```

for some given value of **fn** to populate **divs** with all the **Div** nodes in **tree** that satisfy **fn**. An example of **fn** is

```
let fn (div:SpanXMLDiv) = div.SetlDate > 20100301
```

With this solution it is still necessary to copy and edit the **XNode** function for each of the types in the tree but this is a simpler piece of code which does nothing more than return a success value or **None**, a reasonable compromise.

Finally we present the boiler-plate code to populate one of the **SpanXMLxxx** records, specifically the **SpanXMLClearingOrg**. The steps are simple. We initialise a dictionary which records the state (incomplete, for the most part) of the current record being created. Therefore, this dictionary contains an entry for each of the fields in the record, i.e. each of the simple XML elements contained within the **clearingOrg** XML element. Next we define a function, **read**, which transforms the element into a field in the record. The simple elements are read in directly through an appropriate conversion. Again, this could probably be performed using LINQ-to-XML in the manner demonstrated in [5], especially as we are using a dictionary to store the state, but we will persist with the recursive solution for now.

The complex elements are read in using their own equivalent read function and prepended to the state. Note that there are, in principle, two separate vehicles for retaining the state information; the dictionary already discussed for the simple types and a list for each of the complex types. Having read the **clearingOrg** element and its constituent parts we then construct the **SpanXMLClearingOrg** record setting the fields accordingly and concatenating all the lists of complex XML elements together into a single list, the list of subtrees.

Listing 10

```
let readClearingOrg (reader:System.Xml.XmlReader)
  let dict = ["ec","":>obj; "name","":>obj; ] |> toDict
  let rec read curConv pbRateDef =
    match reader.Name with
    | "ec" as name ->
      dict.[name] <- readAsString reader ; read curConv pbRateDef
    | "name" as name ->
      dict.[name] <- readAsString reader ; read curConv pbRateDef
    | "curConv" as name ->
      read (Node (readCurConv reader) :: curConv) pbRateDef
    | "pbRateDef" as name ->
      read curConv (Node (readPbRateDef reader) :: pbRateDef)
    | _ -> curConv, pbRateDef

  reader.ReadStartElement()
  let curConv, pbRateDef = read [] []
  reader.ReadEndElement()
  SpanXMLClearingOrg(
    {
      Ec = dict.["ec"] :?> string
      Name = dict.["name"] :?> string
    }), (curConv @ pbRateDef)
```

Given equivalent definitions of **readCurConv** and **readPbRateDef** we can write Listing 10.

And there we have it. A lightning-fast discussion of n-ary trees followed by a somewhat more long-winded, yet still abbreviated, example of one in action in the Real World [10].

It's not all work, work, work [11] though. Trees have other uses. For example, one could have written Colossal Cave [12] using a tree structure. Suppose we wanted to recreate something like the 'maze of twisty little passages, all alike' or, indeed, the 'maze of twisty little passages, all different'.

First we need to design the tree structure. We might choose the mutually-recursive

```
type tree =
  | Corridor of int * int * room list
  | DeadEnd
  | Exit
and room =
  | Room of int * tree list
```

The integers would be references into simple arrays of adjectives, so that the description of the nodes in the tree can be varied.

This tree does not directly support cyclic data. To do that with the above structure it would be necessary to use generator functions and slightly redefine the **Corridor** and **Room** to refer to delayed objects. In such a way, a previous state could be substituted for a new node in the tree.

Team Chat

Chris Oldwood considers the benefits of social media in the workplace.

As I write this MSN Messenger is taking its last few breaths before Microsoft confine it to history. Now that they own Skype they have two competing products and I guess one has to go. I'm sad to see it be retired because it was the first instant messaging product I used to communicate with work colleagues whilst I was both in and out of the office.

At my first programming job back in the early 90s the company used Pegasus Mail for email as they were running Novell NetWare. They also used TelePathy (a DOS based OLR) to host some in-house forums and act as a bridge to the online worlds of CompuServe, CIX, etc. Back then I hardly knew anyone with an email address and it was a small company so I barely got any traffic. The conferencing system (more affectionately known as TP) on the other hand was a great way to 'chat' with my work colleagues in a more asynchronous fashion. Although some of the conversations were social in nature, having access to the technical online forums was an essential developer aid. Even the business (a small software house) used it occasionally, such as to 'connect' the marketing and development teams. Whereas email was used in a closed, point-to-point manner, the more open chat system allowed for the serendipitous water-cooler moments through the process of eavesdropping.

As the Internet took off the landscape changed dramatically with the classic dial-up conferencing systems and bulletin boards (BBS's) trying to survive the barrage of web based forums and ubiquitous access to the Usenet. Although I did a couple of contracts at large corporations I still had little use for office email and that continued when I joined a small finance company around the turn of the millennium.

It was nice being back in a small company – working with other fathers who also had a desire to actually spend time with their families – because it meant we could set up remote working. The remote access was VPN based (rather than remote desktop) which meant that we would have to configure Outlook locally to talk to the Exchange server in the office. This was somewhat harder back then and it was just another memory hog to have cluttering up your task bar. A few of us had been playing with this new MSN Messenger thing, which, because we were signed up personally meant that whether we were at home or in the office we easily talk to each other. Given our desire to distribute our working hours in a more family friendly manner that meant we often found ourselves working (remotely) alongside a team-mate in the evening.

we often found ourselves working (remotely) alongside a team-mate in the evening

Instant messaging soon became an integral part of how the team communicated. With the likelihood that at least one of us was working from home we could still discuss most problems when needed. Of course there was always the option to pick up the old fashioned telephone if the limited bandwidth became an issue or the emoticon count reached epidemic proportions. Even 3- and 4-way conversations seemed to work quite painlessly. However shared desktops and whiteboards felt more like pulling teeth, even over a massive 128 Kbps broadband connection.

Eventually I had to move and I ended up back at one of those big corporations – one that was the complete opposite of my predecessor. Here

CHRIS OLDWOOD

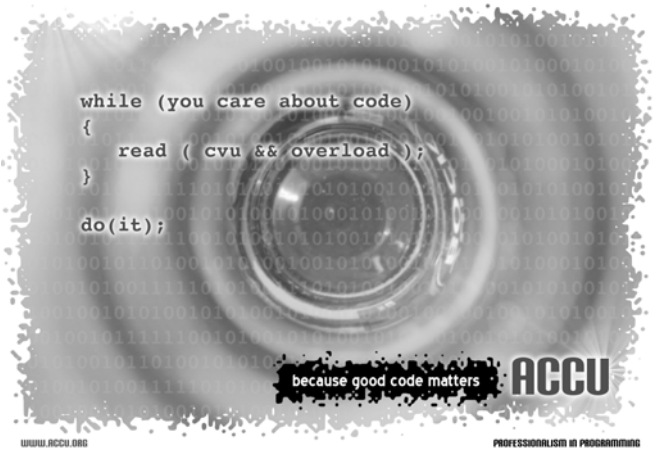
Chris started as a bedroom coder in the 80s, writing assember on 8-bit micros. Now it's C++ and C# on WIndows. He is the commentator for the Godmanchester Gala Day Duck Race and can be reached at gort@cix.co.uk or @chrisoldwood

Let's Talk About Trees (continued)

This, however, we leave as an exercise for the reader, particularly the reader who feels that they ought to contribute an article to **CVu** but just can't think of a topic. ■

References

[1] Tree structures http://en.wikipedia.org/wiki/Tree_(data_structure)
[2] Traversing trees http://en.wikipedia.org/wiki/Tree_traversal
[3] Algebraic Data Type http://en.wikipedia.org/wiki/Algebraic_data_type
[4] ASX Risk Parameter file http://www.asx.com.au/sfe/span.htm
[5] Linq-to-XML example http://stackoverflow.com/questions/9719526/seq-todictionary
[6] Recursive functions using the Accumulator pattern http://htdp.org/2003-09-26/Book/curriculum-Z-H-39.html
[7] Expert F# v2.0, Don Syme
[8] Functional C# http://code.google.com/p/functional-utils-csharp
[9] Span parser on Google Code http://code.google.com/p/span-for-margin/
[10] http://www.youtube.com/watch?v=tjHOk77d4po
[11] http://idioms.thefreedictionary.com/All+work+and+no+play+makes+Jack+a+dull+boy
[12] Colossal Cave walkthrough http://www.ir.bbn.com/~bschwart/adventure.html



everything was blocked, you couldn't (or shouldn't) install anything without approval and instant messaging was blocked by the company firewall. In this organisation email ruled. This was not really surprising because The Business, development teams, infrastructure teams, etc. were all physically separated. Consequently emails would grow and grow like a snowball as they acquired more recipients, questions and replies until eventually they would finally die (probably under their own weight) and just clog up the backup tapes. The company's technical forums were also run using email distribution lists. Anyone brave enough to post a question had to consider the value of potentially getting an answer versus spending the next 20 minutes dealing with the deluge of Out of Office replies from the absent forum participants. They even had a special 'Reply All' plug-in that would pop up a message box to check if you were really, really, really sure that every recipient you were intending to spam actually needed to see your finest display of English prose and vast knowledge of the subject matter.

we often found ourselves working (remotely) alongside a team-mate in the evening

Little known to most employees the company actually ran an internal IRC style chat service. Presumably, in an attempt to reduce the pummelling the Exchange Server was taking, they forced their developers to 'discover' it by making the chat client start up every time they logged in. They also disbanded the email distribution lists and set up IRC channels instead. Even the ACCU had its own channel!

It may sound like a draconian tactic, but it worked, and I for one am really glad they did. Suddenly the heydays of the conferencing system I had used back in the beginning were available once more. Although there was a 'miscellaneous' topic where a little social chit-chat went on I'd say that by-and-large the vast majority of the public traffic was work related. Both junior and senior developers could easily get help from other employees on a range of technical subjects covering tools and languages. Naturally, given the tighter feedback loop, the conversations easily escalated to the level of 'what problem are you trying to solve exactly?' which is often where the real answer lies.

One particular channel was set up to try and enable more cross pollination of internal libraries and tools. In an organisation of their size I would dread to think how many logging libraries and thread pools had been implemented by different teams over the years. Our system also had its own dedicated channel too which made communicating with our off-shore teams less reliant on email. Given the number of development branches and test environments in use this was a blessing that kept the inbox level sane. The service recorded all conversations, which I'm sure to some degree kept the chatter honest, but more importantly transcripts were available via a search engine which made FAQs easier to handle.

When it came time to move contracts once more I was sorely disappointed to find myself back where I was originally with the last company. Actually it was worse because there were no internal discussion lists either that I could find. Determined not to let my inbox get spammed with pointless chatter I set up a simple IRC server for our team to use. My desire was to sell its benefits to other teams and perhaps even get some communities going, even if we had to continue hosting the server ourselves. Internally the company had Office Communicator (OC), which in the intervening years had acquired the same chat product my previous client used, but sadly this extra add-on was never rolled out and so we remained with our simple IRC setup. Contact with some of the support teams was occasionally via OC but email still remained dominant.

For me IRC style communication has been perfect for the more mundane stuff. For example things like owning up to a build break, messing with a test environment, forwarding links to interesting blog posts or just polling to see if anyone is up for coffee. Using a persistent chat service (or enabling client side logging) also allows it be used as a record of events which can be particularly useful when diagnosing a production problem.

I suspect that from a company's perspective they are worried that such as service will be abused and used for 'social networking' instead, which is probably why they blocks sites like Twitter and Facebook. However, if teams are left to their own devices they will fill the void anyway and so a company is better off providing their own service which everyone expects will be monitored and so will probably self-regulate. But the biggest benefit must surely come from the sharing of knowledge in both the technical and problem domains. As the old saying goes, "A rising tide lifts all boats." ■

Write for us!

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

What do you have to contribute?

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

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

Standards Report

Mark Radford looks at some features of the next C++ Standard.

In my last few standards reports I've been going on about the forthcoming ISO C++ standards meeting in Bristol. Well, it is forthcoming no longer and is currently (at the time of writing) taking place. The delegates number about 100 (which is very much on the high side, although not all of them are there every day) and, whereas meetings have traditionally lasted five days, they are now extended with Bristol being the first six day meeting. The pre-meeting mailing contained 96 papers (compare with 41 and 71 papers in the pre-meeting mailings for the early and late 2012 meetings, respectively). Given that the meeting is in the UK for the first time in six years I was disappointed that, because of work commitments, I was unable to attend. However I managed to visit on Wednesday evening, which was a good time to be there owing to the Concepts Lite presentation which I will talk about below.

In November 2012's *CVu* I gave a summary of the structure of the committee: at the time there were three working groups and six study groups. Since then activity has increased so that there are now four working groups and eleven (!) study groups. In addition to the traditional Core, Library and Evolution groups, there is now a separate Library Evolution working group. The list of study groups now consists of: Concurrency and Parallelism (SG1), Modules (SG2), File System (SG3), Networking (SG4), Transactional Memory (SG5), Numerics (SG6), Reflection (SG7), Concepts (SG8), Ranges (SG9), Feature Test (SG10), Database Access (SG11). Note that not all study groups meet daily during the week of the meeting. For example, the Database group (tasked with 'creating a document that specifies a C++ library for accessing databases') only had its first meeting on Thursday morning.

Before going any further I'd like to talk briefly about one of the deliverables a standards committee can produce: that is, a technical specification, or TS for short. Readers may have come across a technical report (TR) before, such as TR1 which proposed various extensions to the library for C++0x. A TR is informational whereas, by contrast, a TS is normative. More information about this can be found on ISO's web site [1].

Readers will no doubt be aware of the Concepts proposal and its troubled journey through the process leading to the C++11 standard, only to be pulled at the eleventh hour. The story of Concepts, in my opinion, should serve as a cautionary warning: the original proposal inspired more ideas and the whole thing grew and grew in complexity. In the end, its removal from the C++11 (C++0x at the time) standard was a pragmatic necessity in order to ship the new standard that had become long overdue.

Now, Concepts are back on the agenda for the future of C++, reinvented in the form of Concepts Lite. The current main source of information on Concepts Lite is the paper 'Concepts Lite: Constraining Templates with Predicates' by Andrew Sutton, Bjarne Stroustrup, Gabriel Dos Reis (N3580). There is also a web site [2]. Given the history of this feature (alluded to above), I had concerns about its reintroduction. Therefore, I was glad I had the chance to go to the Wednesday evening presentation given by Andrew Sutton. This was the same presentation he gave at the ACCU conference and it can downloaded [3]. I found myself liking Concepts Lite. My original understanding was (and I can't remember where it came from) that the aim was for the feature to be in C++14. However, this matter came up at the presentation and Andrew Sutton said this wasn't going to happen, rather there would be a TS instead. Currently there are no library proposals, but the TS will probably include some library features (or there may even be a separate TS for a constrained library). This proposal has generated a lot of interest among the committee, and I expect it will so do so among the C++ community in general. Therefore, I will spend the rest of this report on it, and go into some more detail.

MARK RADFORD

Mark Radford has been developing software for twenty-five years, and has been a member of the BSI C++ Panel for fourteen of them. His interests are mainly in C++, C# and Python. He can be contacted at mark@twonine.co.uk

Concepts Lite

Concepts Lite offer an effective approach to constraining template arguments without the complexity of the original Concepts. They do, however, leave open a migration path to full Concepts. Currently though, they are much simpler than Concepts were. In particular, there is no attempt to check the definition of the template: the constraints are checked only at the point of use. This is a big difference when compared to the Concepts originally proposed: Concepts Lite are intended to check the use – and not the definition – of templates. Other good points include: observed compile-time gains of between 15% and 25% (according to Andrew Sutton), templates can be overloaded on their constraints, and the constraint check is syntactic only. That last point is another source of simplification. Consider an `Equality_Comparable` constraint: this would enable the compiler to check that a template argument type is comparable using `operator==`, but there is no mechanism for attempting to evaluate whether or not the `operator==` has the correct semantics. Regarding overloading, function templates would be selected on the basis that the more constrained template is the better match.

The icing on the cake is that much of Concepts Lite has been implemented on a branch of GCC 4.8 in an experimental prototype [3].

That wraps up another standards report. As usual, N3580 and all the other submitted papers can be found on the website [4]. Finally, I would like to thank Steve Love for his flexibility with deadlines.

References

[1] http://www.iso.org/iso/home/standards_development/deliverables-all.htm?type=ts
[2] http://concepts.axiomatics.org/~ans
[3] http://concepts.axiomatics.org/~ans/accu13.pdf
[4] http://www.open-std.org/jtc1/sc22/wg21/docs/papers

Code Critique Competition 81

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 have been starting to use IPv6 and have tried to write a routine to print abbreviated IPv6 addresses following the proposed rules in RFC 5952. It's quite hard – especially the rules for removing consecutive zeroes. Can you check it is right and is there a more elegant way to do it?

Here is a summary of the rules:

Rule 1. Suppress leading zeros in each 16bit number

Rule 2. Use the symbol "::" to replace consecutive zeroes. For example, 2001:db8:0:0:0:0:2:1 must be shortened to 2001:db8::2:1. If there is more than one sequence of zeroes shorten the longest sequence – if there are two such longest sequences shorten the first of them.

Rule 3. Use lower case hex digits.

The code is in Listing 1.

Listing 1

```
/* cc80.h */
#include <iosfwd>
void printIPv6(std::ostream & os,
  unsigned short const addr[8]);
/* cc80.cpp */
#include "cc80.h"
#include <iostream>
#include <sstream>

namespace
{
  // compress first sequence matching 'zeros'
  // return true if found
  bool compress(std::string & buffer,
    char const *zeros)
  {
    std::string::size_type len =
      strlen(zeros);
    std::string::size_type pos =
      buffer.find(zeros);
    if (pos != std::string::npos)
    {
      buffer.replace(pos, len, "::");
      return true;
    }
    return false;
  }
}
void printIPv6(std::ostream & os,
  unsigned short const addr[8])
{
  std::stringstream ss;
  ss << std::hex << std::nouppercase;
  for (int idx = 0; idx != 8; idx++)
  {
    if(idx) ss << ':';
    ss << addr[idx];
  }
  // might be spare colons either side of
  // the compressed set
  while (compress(buffer, ":::"))
    ;
  os << buffer;
}

/* testcc80.cpp */
#include <iostream>
#include <sstream>
#include "cc80.h"
struct testcase
{
  unsigned short address[8];
  char const *expected;
} testcases[] =
{
  { {0,0,0,0,0,0,0,0},
    "::" },
  { {0,0,0,0,0,0,0,1},
    "::1" },
  { {0x2001,0xdb8,0,0,0,0xff00,0x42,0x8329},
    "2001:db8::ff00:42:8329" },
};

#define MAX_CASES sizeof(testcases) /
sizeof(testcases[0])

int test(testcase const & testcase)
{
  std::stringstream ss;
  printIPv6(ss, testcase.address);
  if (ss.str() == testcase.expected)
  {
    return 0;
  }
  std::cout << "Fail: expected: "
    << testcase.expected
    << ", actual: " << ss.str() << std::endl;
  return 1;
}

int main()
{
  int failures(0);
  for (int idx = 0; idx != MAX_CASES; ++idx)
  {
    failures += test(testcases[idx]);
  }
  return failures;
}
```

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

Critiques

I obviously failed to produce an interesting enough example this time as nobody wrote a critique. That may of course be because few readers are interested in IPV6: I believe the readership of this magazine mostly comes from countries where the shortage of IPV4 addresses is not yet a serious problem. Take-up of IPV6 is most prevalent in countries where the use of the Internet is developing rapidly, such as India and China. Either that, or nobody thought there were any problems with the code.

Commentary

The first trouble with the code above is the use of an array of 8 short integers to represent an IPV6 address. There may be problems with network byte ordering if the IP addresses used as examples are passed unchanged to a network call. It would be a lot better to use the standard data structures for IP addresses such as in this case **`in6_addr`**.

It is surprisingly hard to print out (or read in) IPV6 addresses by hand. Fortunately there are very few cases when this is advisable – using a standard facility is very strongly recommended.

We do not at present have such a facility in C++ although the networking study group is discussing proposals for a network address class or classes; if a consensus is reached and adopted we might have a standard C++ way to do this before too long. In the meantime you could use boost (**`boost::asio::ip::address`**): see the **`to_string`** method of that class.

The function **`inet_ntop`** is one standard way to do this in C.

```
char dst[INET6_ADDRSTRLEN];
if (!inet_ntop(AF_INET6, addr,
  dst, sizeof(dst)))
{
  // handle error...
}
```

I would probably try to avoid critiquing the user's code as provided and focus their attention on using a standard facility.

However, once this has been accomplished, I might return to their code and point out that searching the string for **`"0:…"`** incorrectly matches the initial zero against any hex number with a trailing zero digit. The test cases provided by the user failed to cover this case.

Such failings in test coverage are quite common. For example, a bug was discovered with the streaming of doubles in Visual Studio 2012 (http://connect.microsoft.com/VisualStudio/feedback/details/778982). I am sure Microsoft have test coverage of this operation; but obviously their test data set lacked adequate coverage.

The trouble with doing the abbreviation of the longest run of zeros with the textual representation is that there are boundary conditions at both the beginning and end of the string. I think the easiest algorithm is to pass through the binary representation to find the start address and length of the longest run and then use this information when converting the representation to characters.

The algorithm though misses another special case – that of IPV4 mapped and compatible addresses. These have an alternative convention for display which emphases the IPV4 'nature' of the address. So, for example, the IPV6 address 0:0:0:0:0:ffff:c00:280 would be displayed as ::ffff:192.0.2.128 on many platforms.

This is would hopefully provide another reason to reinforce why using the standard function is normally preferable to writing your own.

Finally I notice that the code to join the eight short integers together with a colon delimiter is addressed by the recent C++ proposal N3594 ('**`std::join()`**: An algorithm for joining a range of elements').

Code Critique 81

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

I am new to C++ and trying to write some objects to disk and read them back in. How can I get the pointer to the objects that are read back in?

Where would you start with trying to help this newcomer?

The code is in Listings 2, 3, 4 and 5 (note: it uses a few C++11 features so will need modifying to run on a non-conformant compiler):

Listing 2

```
/*
 * Bike.h
 */

#ifndef BIKE_H_
#define BIKE_H_

#include <iostream>
#include <string>
#include <vector>
#include <iterator>
#include <algorithm>
#include <iomanip>
#include <ios>

class Bike {
  Bike* address;  // Pointer to Bike object
  std::string name;
  double price;
  std::string make;

public:
  //Bike(); // eliminate to avoid ambiguity
  Bike(Bike* a, const std::string& n =
    "unknown", double p=0.01,
    const std::string& m="garage") :
    address(a), name(n), price(p), make(m){}
  virtual ~Bike();

  inline std::string getName(){return name;}
  inline double getPrice(){return price;}
  inline std::string getMake(){return make;}
  inline Bike* getAddress(){return address;}

  static void writeToDisk(
    std::vector<Bike> &v);
  static void readFromDisk(std::string);
  static void splitSubstring(std::string);
  static void restoreObject(
    std::vector<std::string> &);
};

std::ostream& operator << (std::ostream& os,
  Bike &b);

#endif /* BIKE */
```

```
/*
 * Bike.cpp
 */

#include "Bike.h"

//Bike::Bike() {} // TODO Auto-generated stub

Bike::~Bike() {} // TODO Auto-generated stub

std::ostream& operator << (std::ostream& os,
  Bike &m){
  os << std::left << std::setw(10)
     << m.getAddress() << "\t"
     << m.getName() << "\t"
     << m.getPrice() << "\t" << m.getMake();
  return os;
}
```

Listing 4

```
/*
 * file_io.cpp
 */
#include "Bike.h"
#include <fstream>
#include <iomanip>
#include <iostream>
#include <iterator>
#include <vector>
#include <cstring>
#include <sstream>
#include <algorithm>

// Write objects to disk

void Bike::writeToDisk(std::vector<Bike> &v){
std::ofstream out_2("bike_2.dat");
for (auto b:v){
  out_2 << b.getAddress() << ':'
      << b.getName()
      << ':'<< b.getPrice() << ':'
      << b.getMake()  << std::endl;
  }
out_2.close();
}
//-----------------------------------------
//Read from disk into vector and make objects
void Bike::readFromDisk(
  std::string bdat) // "bike_2.dat"
{
  std::cout << "\nStart reading: \n";
std::vector<char> v2;
std::ifstream in(bdat);
   copy(std::istreambuf_iterator<char>(in),
       std::istreambuf_iterator<char>(),
       std::back_inserter(v2));
in.close();

for(auto a:v2){
   std::cout << a;  // Debug output
   }
std::string s2(&(v2[0])); // Vector in String
std::cout << "\nExtract members:\n";
while (!s2.empty()){
  //  objects separated by \n
  size_t posObj = s2.find_first_of('\n');
  std::string substr = s2.substr(0,posObj);
  s2=s2.substr(posObj+1);
  splitSubstring(substr);
  }
}
void Bike::splitSubstring (std::string t){
  // Save the address and the members in v3
std::vector<std::string> v3{(4)};
size_t posM; // [in substring]
int i;
for (i=0; i<4; i++){
  posM = t.find_first_of(':');
  v3[i] = t.substr(0,posM);
  if (posM==std::string::npos) break;
  t=t.substr(posM+1);
  }
for(auto member:v3){
  std::cout << std::setw(10) << std::left
    << member << "   \t";}
  restoreObject(v3);
  std::cout << std::endl;
  v3.clear();
}
```

Listing 4 (cont'd)

```
void Bike::restoreObject(std::vector<std::string>
&v3){
  Bike* target;  // I want the object here ...
  double p;
  std::stringstream ss(v3[2]);
  ss >> p;
  Bike dummy{&(dummy),v3[1], p, v3[3]};
  target =  &(dummy);
  std::cout << "\nRestore: " << *target
    << std::endl;
  }
```

Listing 5

```
/*
 * main_program.cpp
 */

#include "Bike.h"
#include <fstream>
#include <iomanip>
#include <iostream>
#include <iterator>
#include <vector>
#include <cstring>
#include <sstream>
#include <algorithm>

int main(){
  std::cout << "start\n";
  std::vector<Bike> v;
  Bike thruxton{&(thruxton), "Thruxton",
    100.00 , "Triumph"};
  Bike sanya{&(sanya)};
  Bike camino{&(camino), "Camino   ",
    150.00, "Honda"};
  Bike vespa{&(vespa), "Vespa    ",
    295.00, "Piaggio"};

  v.push_back(thruxton);
  v.push_back(sanya);
  v.push_back(camino);
  v.push_back(vespa);

  for(Bike b:v) std::cout << b << std::endl;
  // using overloaded << operator

  Bike::writeToDisk(v);
  // restore objects
  Bike::readFromDisk("bike_2.dat");
  // where are the restored objects??
  return 0;
}
```

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.

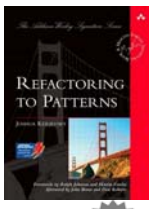
If you want to review a book, your first port of call should be the members section of the ACCU website, which contains a list of all of the books currently available. If there is something that you want to review, but can't find on there, just ask. It is possible that we can get hold of it.

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-glamorous 'not recommended' rating, you are entitled to another book completely free.

Thanks to Pearson and Computer Bookshop for their continued support in providing us with books.
Jez Higgins (publicity@accu.org)

Patterns

Refactoring to Patterns



By Joshua Kerievsky, published by Addison Wesley ISBN: 978-032121335

Reviewed by Alan Lenton

For some reason this book escaped my notice until recently, which is a pity, because it's a very useful book indeed. Quite a lot of programmers, even those using agile methods, seem to think that patterns are merely something that you spot at the design stage. This is not the case, though it's useful if you do spot a pattern early on. Programs evolve, and as they do, patterns become more obvious, and indeed may not have been appropriate at earlier stages of the evolution.

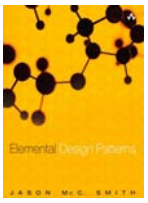
The book, as its title implies, deals with evolving programs, and does it very well. The bulk of the book takes a relatively small number of patterns and, using real world examples, gives a step by step analysis, with Java code, of how to refactor into the pattern. As long as readers do treat these as examples, rather than something set in stone, they will learn a lot about the arts of identifying patterns and the nitty gritty of refactoring.

I also liked the pragmatism of the author. Unlike some pattern freaks, he freely admits that there are times when using a specific pattern is overkill, especially where the problem is simple. Most people, myself included, when the idea of patterns are first grasped, tend to see patterns in everything and immediately implement them. This is frequently inappropriate, and rather than making the program structure clearer, muddies the waters. There are a number of warnings in the book against this approach.

I was very impressed by this book. In fact it is one of a small number of books that has made it to my work desk, where it fits, both intellectually and literally, in between the Gang of Four's *Design Patterns*, and Martin Fowler's *Refactoring!*

Highly recommended.

Elemental Design Patterns



By Jason McC. Smith, published by Addison-Wesley ISBN: 978-0321711922

Reviewed by Alan Lenton

This is an interesting book, well researched and well written. Its basic thesis is that the better known design patters, such as those explored by the 'Gang of Four', can be decomposed into more elementary patterns, and so on, until we have a number of elemental patterns which cannot be broken down any further.

As part of this process, the author introduces a new type of diagram which he calls 'PIN' – Pattern Instance Notation. Since PIN is used extensively in the book to represent patterns it is essential to understand PIN fully. Unfortunately, there don't seem to be any tools that allow you to draw the PIN symbols, so it's difficult to learn it by using the system. This is something of a weakness in the book.

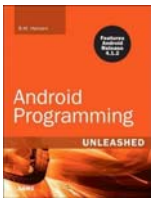
That problem aside, the book is an interesting exposition of the fundamentals of patterns. However, the patterns Smith introduces are so very basic – Create Object, Recursion, and Inheritance, for instance – that I doubt that most application programmers will find its elementary pattern catalog particularly useful on a day to day basis. However, anyone involved in designing and programming refactoring browsers or the refactoring elements of an IDE will find the contents of the book very useful.

As a tool for automatically identifying incipient patterns in existing code I haven't seen anything that comes near it. Indeed, this was the genesis of the book – a project to automate the identification of certain patterns. Object oriented language designers may also find it useful for figuring out what they might need to build into their languages. The academic nature of the book is emphasized by substantial section on the formal logic involved using p-Calculus, on which I don't feel qualified to comment.

Overall, I'd say that this may well be a useful book if you want to study patterns in more depth, or you are interested in automatic pattern recognition. As an ordinary programmer, though, you won't find a great deal that's of instant use, since the patterns described are of a sufficiently low level that they are built into the language and idiom of most object orientated languages.

Android

Android Programming Unleashed



By B.M. Harwan, published by Sams

Reviewed by Paul F. Johnson

Not recommended – not even slightly recommended unless you like levelling up beds and even then, I can think of better books.

This is my first review in what seems an eternity and unfortunately, it's not a good one.

The Android market it getting bigger by the minute and with that, more and more books are coming out professing to show you how, in 200 pages, you can go from a user to someone who can create an app that redefines the landscape for apps out there. This is no exception.

It starts by wasting the first chapter telling you how to install the Android SDK. Why? The installer pretty much does everything for you now. Sure you may need to know how to set up the emulators and you may wish to not just accept the defaults, but why waste a chapter on it? That said, I have the same issue with most books of this ilk; "let's use a chapter to show some screen dumps of how to install Visual Studio". Just annoying.

Okay, that bit over. What's left? Code errors everywhere, poor explanations of how things work and why they're done like that and did I mention stuff that plain doesn't compile? No? There is quite a bit of it.

Ok, let's look at a particular example on page 188. A nice simple media player.

```
public class PlayAudioAppActivity
extends Activity {
@override
```

```
public void onCreate(Bundle
savedInstanceState) {
super.onCreate(savedInstance);
setContentView(R.layout.activity_
play_audio_app);
Button playButton =
(Button)findViewById(R.id.playbtn
);
playButton.setOnClickListener(new
Button.onClickListener() {
public void onClick(View v) {
MediaPlayer mp =
MediaPlayer.create(PlayAudioAppAc
tivity.this, R.raw.song1);
mp.Start();
}
});
```

Looks ok, except for the 2 lines that do anything – (`MediaPlayer mp =` ... and `mp.Start()`)

What's it doing? That's right, it creates an instance using the Activity context and then uses a file in the `raw` directory. What `raw` directory? Unless the author created one, there isn't one. It then starts the media player. No sort of trapping or defensive coding, just hey, it works or I'll leave you to puzzle out why it didn't – and given quite a lot of the code is written in the same way, the poor ol' user is left scratching their head and wondering why SAMs didn't include a source code CD or not limit getting the code for 30 days past purchase of the book.

Now, let's add some insult to injury. The book claims to cover up to 4.1.2, so let's look at something that varies massively over the versions – animation. Prior to version 3, animation was pretty awful. It worked, but wasn't really that good.

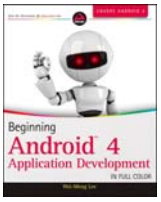
Google rewrote huge chunks for version 3 and animation was there and happy. What does this book say about the differences? Nothing. It sticks to what is there prior to version 3. The only saving grace is the section on using animation using XML.

Android comes with SQLite databases as standard. Why then does the author go about creating a custom database using ArrayLists?

I could really rip into this book and I mean seriously rip into it, but at the end of the day life is too short to waste my time trying to find code in there that works as it should.

Given the author is also a time-served lecturer with an 'easy to understand' style, I'm amazed this managed to get past the technical editor's eye – unless said technical editor is one of his students...

Beginning Android 4 Application Development



By Wei-Meng Lee, published by Wrox, ISBN 978-1-118-19954-1

Reviewed by Paul F. Johnson

Recommended with reservations.

There are plenty of awful Android books out there (see my other review for one such example). Lots of errors in the code, broken examples, wasted paper, illogical layouts and well, pretty much a waste of a tree. This is NOT one of those books.

This is a rather good book. Not amazing, but still far better than a lot of things out there. From the word go, there are screen shots a-plenty, lots of code examples with the emphasis definitely on trying things out for yourself. But therein lies the problem with the book. It is all well and good having example code, but not when you have to disappear onto a website and dig around for it (it is why this review is on Recommended and not Highly Recommended).

A major omission is the lack of anything on graphics handling. While it does show you how to display graphics, there is nothing on drawing or use of the camera. An omission which while understandable, does detract from this book quite a lot. Drawing leads into long and short presses, drags, canvases and other fun bits and pieces. Perhaps for the next edition this could be included? Here's hoping!

The author of this work does know what he is on about with a clear way to his writing style.

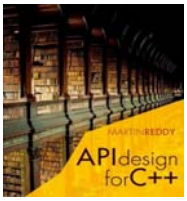
I will happily admit that I don't do Java. I've never understood it and really, it doesn't make too much sense to me. I do, however, program for Android using Xamarin.Android (or Monodroid as it was). There is only one or two books out there that are dedicated to using .NET on Android. The beauty of this book though is that it explains how the system works and how events are used and as long as you know the equivalent in .NET world, this book provides you with a great resource that is currently missing.

The book covers just about all of the main parts of Android development (including data persistence, maps, messaging and networking) up to Ice Cream Sandwich. Jellybean doesn't appear to be in the book.

All in all, this is one of the better books out there for Android development. It's good, but has its failings.

Miscellaneous

API Design for C++



By Martin Reddy, published by Morgan Kaufmann ISBN: 978-0123850034

Reviewed by Alan Lenton

Martin Reddy has written a very useful book on the art and science of Application Programming Interfaces (APIs), and along the way has produced a book chock full of useful hints and help for more junior programmers. It is not a book for someone wanting to learn to program in C++, but if you have been programming in C++ for a year or so, then you will find this book will help you move toward towards program design instead of just 'coding'.

Obviously, the book concentrates on API design, but along the way it covers selected patterns, API styles, performance, testing and documentation. As a bonus it also covers scripting and extensibility, and I found the section on plugins particularly useful. An appendix covers the varied technical issues involved in building both static and dynamic libraries on Windows Mac and Linux.

The only minor disagreement I would have with the author is with the extent to which he goes to move internal details out of header files in the name of preventing the API users from doing anything that might allow them to access those features. From my point of view, using the API is a type of contract between the API writer and the API user. If the user is foolish enough to break that contract then he or she has to take the consequences in terms of broken code when a new version of the library comes out. In any case this sort of behaviour should be picked up by code review in any halfway decent software studio.

That is, however, a minor niggle, and this book represents a rich seam for programmers to mine for good programming practices – even if you aren't writing API, your use of them will improve dramatically!

The Essential Guide To HTML5: Using Games To Learn HTML5 And JavaScript (Paperback)

By Jeanine Meyer, published by FRIENDS OF ED ISBN: 978-1430233831

Reviewed by Alan Lenton

I really can't recommend buying this book. It seems to have been written mainly for people with a very short attention span, and therefore skips on explaining why you do things in a specific way. The chosen way of displaying programs listings, while it might have be useful for annotating each line, makes it impossible to look at the program flow, or consider the over all design. The one correct idea – that of incremental program development – becomes merely a vehicle for large spaced out repetitive chunks of code which probably extend the size of the book by as much as 20%.

The code itself, is, how shall I put it, somewhat less than optimal, and not conducive to creating good coding habits by those learning from the book. For instance, in the dice game example, the code for drawing a dot on the dice is repeated in a 'cut and paste' style every time a dot is drawn, instead of being gathered into a function and called each time it is needed.

I shudder to think about what sort of web site someone who learned from this book would put together. Fortunately, perhaps, they are not likely to learn enough from the book to make a web site work.

A triumph of enthusiasm over pedagogy. Definitely not recommended!

View from the Chair

Alan Griffiths
chair@accu.org

I've been given special dispensation by the *C Vu* editor to submit this report 'late' (that is, after the conference and AGM). I know I enjoyed the conference and believe that most of those who could attend did so too. This was the first year away from Oxford, from what I saw it was a mostly successful move. There were a few 'opportunities for improvement' but I'm sure that the conference committee will be considering carefully what lessons can be applied for next year.

As this is written after the AGM it is possible to report on proceedings there. We had a number of votes and proxies registered on the motions for constitutional change before the meeting, but both these and the votes at the meeting were overwhelmingly in favour of the proposed changes.

There were two constitutional motions passed: one proposed by Roger Orr and Ewan Milne to rationalise the committee posts required by the constitution; and, a much larger one proposed by Giovanni Asproni and Mick Brooks to support voting by members that cannot attend the AGM.

Last year's AGM made changes to the constitution that required constitutional motions be notified in advance and that preregistered and proxy votes on these motions be accepted. There was also a call for the committee to be more transparent about the way the organisation runs.

In line with this we've used the members mailing list to prepare the proposed changes to the constitution. Drafts of these motions were posted to the list and updated in response to comments: this meant that issues could be addressed in advance of the AGM and the final wording we had was passed quickly.

The committee has been taking other steps over the year to make the operation of the organisation more transparent. As part of this minutes of the committee meetings are now published on the accu-members mailing list once they are approved. Also, while committee meetings have always been open to members (subject to prior arrangement with the secretary) they can now be attended remotely.

At this year's AGM there was also a call from the floor to ensure that committee members from overseas could attend committee meetings. This didn't move to a vote as the same technology that allows members to attend remotely is already in use by committee members.

One notable failure by the committee was that we didn't have the accounts ready for the AGM. This has happened before, but was unexpected: the treasurer got the figures to the accountant in what we believed was good time and we only realised that the accounts going to be late when they didn't appear as expected. In the end, the accounts were actually available for the AGM, but no-one present (neither committee members, nor the honorary auditors) had had a chance to review them. We will be investigating further at the next committee meeting.

As anticipated by my last report we've had a couple of people stand down from the committee – Mick Brooks has been replaced as Membership Secretary by Craig Henderson. Tom Sedge and Stewart Brodie have also stepped down.

My last report also mentioned the 'hardship fund'. This was originally created to support the memberships of individuals who could not finance themselves. However, there have been decreasing calls on this fund over the years and while it is still possible to donate, nothing has been paid out for quite some time. The committee needs guidance on how to proceed: Should we continue accepting contributions to the fund? And what do we do with the money already donated?

We do sometimes offer concessionary memberships to members in financial difficulties. So one option for using the hardship fund could be to 'make up' the difference (effectively transferring the money to our general budget). It would also be possible to spend the money in new ways (supporting attendance at the Conference for example). If you can suggest something else then the committee would be pleased to consider it.

We still have no volunteer to moderate the accu-contacts mailing list. This isn't an onerous task (there are a few emails each week to classify as 'OK', 'Spam' or 'needs fixing') but we don't currently have a replacement for this role. Is this something you could do for the ACCU?

Please contact me at chair@accu.org about any of the items above.

Letter to the Editor

Dear Editor,

Not having used F# before, it was interesting to read Richard Polton's article, 'Comparing Algorithms', in the March 2013 *C Vu*, in which he compares a variety of iterative and recursive solutions to the first Project Euler problem (sum the multiples of 3 or 5 below 1000).

As a didactic exercise, this is all well and good. The problem itself, though, strikes me as being very similar to the one in the famous story about Carl Friedrich Gauss; and indeed it turns out that the sum of all multiples of *i* less than or equal to *n* can be found (in plain old C) without any loop at all:

```
int sum(int i, int n) {
  return i * (n/i * (n/i + 1))/2; }
```

We can use this to find the sums of the multiples of 3 and of 5, then remove the ones we've double-counted:

```
int euler1(int i1, int i2, int n) {
  return sum(i1, n-1) + sum(i2, n-1)
 - sum(i1*i2, n-1); }
```

In C++, if the arguments are constants we can convert function arguments to template parameters, reducing our runtime cost all the way to zero by doing the work at compile time; and in C++11 we should be able to accomplish the same thing simply by sticking `constexpr` in front of both functions.

I think the lesson is that, while iteration, recursion, functional programming, templates, C++11, and all our other flashy tools and techniques each have their place, in our eagerness to try out our new capabilities we mustn't lose sight of the problem we originally set out to solve.

Sincerely,

Martin Janzen
(martin.janzen@gmail.com)

If you read something in C Vu that you particularly enjoyed, you disagreed with or that has just made you think, why not put pen to paper (or finger to keyboard) and tell us about it?