The magazine of the ACCU

www.accu.org

{cvu}

Volume 22 • Issue 6 • January 2011 • £3

Features

Regulars

{cvu}
Volume 22 Issue 6
January 2011
ISSN 1354-3164
www.accu.org

Look at the princess

I was originally asked to cover as C Vu editor in October while Steve and Frances were getting married. However, my wife and I were expecting a little boy in July and I couldn't see myself having enough time to do it justice. As it turns out I wouldn't have had anything like enough time as Nathaniel Jacob Grenyer came along, eventually, on the 2nd of August and we've been flat out since. I've only just been able to to shoe-horn in enough time to make a proper job of the January issue as, on top of everything else, I decided to change jobs too. It is no longer a secret that I will be contracting at a financial institution well known to a lot of ACCU members, for six months, with Alan Griffiths and my old boss at Lehman Brothers, Burkhard Kloss.

I have had quite a clear idea of what I wanted to have in my first C Vu as editor for quite some time. I wanted articles from members of the ACCU that I consider to be the big hitters and I wanted it to have a Java feel. In this edition you'll find articles by Kevlin Henney, Jon Jagger and Russel Winder. There are all the usual high quality regulars, such as Pete Goodliffe's 'Becoming a Better Programmer' column and there are two short articles from, to my knowledge, new writers for the ACCU. I know you'll enjoy all of them. What you won't find is anything about Java. You can't have everything!

Until next time....

PAUL GRENYER
GUEST 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 23.1: 1st February 2011
C Vu 23.2: 1st April 2011

Overload 102: 1st March 2011
Overload 103: 1st May 2011

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

COPYRIGHTS AND TRADE MARKS

Sustainable Space

Kevlin Henney shares a code layout pattern.

You are coding in a block-structured language. Spacing is normally used in source code to express logical grouping and separation. For example, indentation is used to show the dependency of code on certain decision points, such as conditional and loop statements.

But what about logical lines of code that are too long to fit on a single physical line? They need to be broken up, but where should the line break appear and how should the overflow be aligned? Simply breaking when the line overflows and using ordinary indentation after that creates an ad hoc layout that appears clumsy and irregular:

```
assignedVariable = firstTerm +
    secondTerm;
targetObject.methodCall(firstArgument,
    secondArgument);
```

What is needed is a regular structure that demonstrates more care in the code and is easier to skim read. It is tempting to break the line after the first term on the right-hand side of an assignment or after the first argument of a method call:

```
assignedVariable = firstTerm +
                   secondTerm;
targetObject.methodCall(firstArgument,
                        secondArgument);
```

This kind of spacing can also apply to method definitions and variable declarations. The alignment makes it easier to see the related terms together. However, the alignment is only local to a call, assignment, variable declaration or method definition. Different expressions or argument lists in the same method or the same class are not at the same alignment, making the overall formatting ragged in appearance. More problematically, however, is the effect of change on this kind of formatting. If any of the names on the first line to the left of the alignment change length, all the spacing becomes misaligned. This style is briefly pleasing, but ultimately brittle. The common task of identifier renaming should not become a high-maintenance obstacle when laying out code.

Therefore, place line breaks somewhere before rather than after the first term of the expression or clause to be broken up. The line following should be indented with respect to the left-hand side of the preceding line. Many formatting styles satisfy this constraint. For example:

```
assignedVariable =
    firstTerm + secondTerm;
targetObject.methodCall(
    firstArgument,
    secondArgument);
```

Or:

```
assignedVariable
    = firstTerm + secondTerm;
targetObject
    .methodCall(
        firstArgument, secondArgument);
```

Spacing no longer depends on the syntax of identifier lengths of the items on the first line. Not only is the spacing independent and, therefore, unaffected by renaming, it also serves to emphasise the 'subject' of each expression more clearly, making the relationship with the operands or terms clearer.

In terms of screen real estate, this style uses less horizontal space and slightly more vertical space than other styles. The code has a slightly more compact appearance and is more consistently placed with respect to the left-hand side of the screen. However, a reader will not see other alignment between lines, such as finding all variable names lined up in the same column. Although such alignment has some local aesthetic appeal to it, it is ultimately not sustainable and should be considered subordinate to maintaining a more global aesthetic and consistency. ■

Acknowledgement

This pattern was first written up and discussed at the Pattern Languages of Tandberg event in Oslo, December 2009. My thanks to those who gave their feedback on the original draft, from which this write-up has evolved. Some discussion of robust layout on accu-general in August 2010 also helped to reinforce a couple of points and highlight a couple of omissions.

KEVLIN HENNEY

Kevlin is an independent consultant and trainer based in Bristol. His development interests are in patterns, programming, practice and process. He is co-author of *A Pattern Language for Distributed Computing* and *On Patterns and Pattern Languages*.

Experiences of Pair Programming

Chris O'Dell shares her experiences of pair programming.

In the last CVu Editorial, Steve asked what pair programming is like as he had had limited experience, usually on one-off tasks lasting no more than an hour at a time and in all cases he's found the required level of concentration to be extremely high – and it is. Three months ago I joined 7digital, a leading digital media delivery company, whose development team use an agile approach including daily stand-ups, Test Driven Development (TDD) [1] and kanban boards [2], plus pair programming. In fact, the whole team pair programs as much of the time as is possible. As Steve found out it can be quite exhausting, and I certainly discovered this after my first couple of weeks with this approach. We hold regular 1-2-1 meetings with the Development Lead and I mentioned this to him in my first meeting. He agreed, pair programming is indeed intensive, and it is counter-productive to be 100% focussed at all times, frequent breaks were recommended and the pair will generally break for 10 minutes returning to their respective machines to catch up on office email, read a little Twitter and some blogs before jumping straight back in ready to focus once again. I would say it is similar to the concentration techniques out there such as Pomodoro [3], whereby you focus solidly for a set amount of time, then break, which is an interesting technique that I have tried in the past and actually found myself achieving more.

One other thing I brought up in my 1-2-1 was that I tended to take a more 'passive' role in the pairing – watching and commenting rather than coding. As I was new to the code base this felt like the right approach to take, but as is always the case, the learning does not sink in until you actually type the code yourself. One method to increase my participation was a 'ping-pong' pair programming [4] approach, whereby one developer writes a failing unit test and the other developer writes the code for it to pass, which fits in excellently with our TDD approach. But, all in all I needed to work on being more assertive. As my Lead had stated, pair programming is a skill just like any other and it must be learned and honed for the benefits to be reaped.

Also on the personal side, pair programming got me up to speed with the code base extremely quickly, and with pairs swapping once or twice a week, I got an overview of many parts of the system far quicker than if I'd been programming alone on a single feature. I also got to know the team which is a massive bonus as there is so much to be learned from your colleagues.

On the business side, we have reported a huge decrease in bugs reaching production and thereby a marked increase in customer satisfaction. The 'second pair of eyes' which your partner provides is hugely valuable – how many times have you found yourself going in circles trying to track a bug down only to find it was a silly typo or something just as obvious? Your pair is your saviour in these situations and they more often than not point out the mistake before it gets to compilation and nowhere near production. The changing of pairs may sound chaotic, but what usually happens is that one person 'owns' the story and partners switch around. The advantage of this is knowledge transfer. Of course, at first there is a small part of needing to bring the new pair up to speed, but then they can jump straight into assisting with development. If it takes too long to bring someone up to speed, then your task is too big or your approach is too unwieldy, either way it needs to be reassessed, which you can do with your pair. This hands-on approach to knowledge transfer is far more effective than with the often arcane 'handover document' usually written just before you head on a three week holiday. As an example there was a piece of work to allow for gift card purchases, broken down into many stories but with one overall Minimum Marketable Feature (MMF) [5] and a colleague *owned* this piece of work with at least 3 other developers having paired with him on it and I was one of them. He took a week's holiday during the development of the MMF and for that week I took over the ownership of the work pairing with one of the other guys and we continued it through to near completion without a hiccup nor a handover document in sight. From a business point of view the work did not need to stop or even slow down noticeably while he was away and the knowledge of the implementation is spread between at least 4 people. From the Dev's point of view he got to take holiday when he wanted and without rushing a handover the afternoon of his last working day.

As with all methodologies it is not a silver bullet and relies heavily on the attitudes of the developers involved. If anyone prefers to work alone – building empires of arcane code, then they will find ways to continue to do so and make their pair miserable in the process. In fact, there's an excellent list of 10 ways to Kill Pair Programming [6], which I suggest you read and do the complete opposite of everything stated.

Your pair is your reviewer, your counsel, your sounding board, your mentor, your student and the person keeping you focussed. There is one thing though that I do miss from coding alone, and that is listening to music with my headphones on, but I can happily trade this for the benefits listed above. ■

References

[1] http://en.wikipedia.org/wiki/Test-driven_development
[2] http://en.wikipedia.org/wiki/Kanban
[3] http://www.pomodorotechnique.com/
[4] http://en.wikipedia.org/wiki/Pair_programming#Ping_pong_pair_programming
[5] http://www.netobjectives.com/glossary/7#letterm
[6] http://www.awkwardcoder.com/index.php/2010/08/27/10-ways-to-kill-pair-programming/

CHRIS O'DELL

Chris is a C# Web Developer working in London at 7digital who is constantly learning. When not programming she can be found with her head in a fantasy book or a manga comic.

When It's Done, It's Done

Pete Goodliffe implores us to stop. When it's time to.

In the name of God, stop a moment,
cease your work, look around you.

Leo Tolstoy

A program is made of a number of subsystems. Each of those subsystems is composed of a smaller parts - components, modules, classes, functions, data types, and the like. Sometimes even boxes and lines. Or clever ideas.

The jobbing programmer moves from one assignment to the next; from one task to another. Their working day is composed of a series of construction and maintenance tasks on a series of these software components: composing new parts, stitching parts together, extending, enhancing or mending existing pieces of code.

So our job is simply a string of lots of smaller jobs. It's recursive. Programmers love that that kind of thing.

Are we there yet?

So there you are: getting the job done. (You think.)

Just like a small child travelling in the back of a car constantly brays 'are we there yet?', pretty soon you'll encounter the braying manager: 'are you done yet?'

This is an important question. It's essential for a software developer to be able to answer that one simple request: to know what 'done' looks like, and to have a realistic idea of how close you are to being 'done'. And then to communicate it.

Many programmers fall short here; it's tempting to just keep hacking away until the task seems complete. They don't have a good grasp on whether they're nearly finished or not. They think: There could be any number of bugs to iron out, or unforeseen problems to trip me up. I can't possibly tell if I'm almost done.

But that's simply not good enough. Usually, avoiding the question is an excuse for lazy practice, a justification for 'coding from the hip', without forethought and planning. It's not methodical.

It's also likely to create problems for you. I often see people working far too hard:

- They are doing more work than necessary, because they didn't know when to stop.
- Without knowing when they'll be done, they don't actually complete the tasks they think are finished. This leads to having to pick things back up later on, to work out what's missing and how to stitch it in. Code construction is far slower and harder this way.
- The wrong bits of code get polished, as the correct goal was never in sight. This is wasted work.
- Developers working too hard are forced to put in extra hours. You'll not get enough sleep!

Let's see how to avoid this and to answer 'are we there yet' effectively.

Developing backwards: decomposition

Different programming shops manage their day-to-day development efforts differently. Often this depends on the size and structure of the software team.

Some place a single developer in charge of a large swathe of functionality, give them a delivery date, and ask them for occasional progress reports. Others follow 'agile' processes, and manage a backlog of more granular tasks (perhaps phrasing them as stories), divvying those out to programmers as they are able to move into a new task.

The first step towards defining 'done' is to know exactly what you're working on. If it's a fiendishly large and complex problem, then it's going to be fiendishly complex to say when you'll be done.

It's a far simpler exercise to answer how far through you are through a small, well-understood problem. Obvious, really.

So if you have been allotted a monster task, before you begin chipping away at it, break it down into smaller, understandable parts. Too many people rush headlong into code or design without taking a step back to consider how they will work through it.

> Split large tasks up into a series of smaller, well-understood tasks. You will be able to judge progress through these more accurately.

Often this isn't a complex a task, at least for a top-level decomposition. (You may have to drill down a few times. Do so. But take note: this is an indication that you've been handed a task at far too high a granularity.)

Sometimes such a decomposition is hard to do, and is a significant task itself. Don't let that put you off. If you don't do it up-front for estimation purposes, you'll only end up doing it later on in less focussed ways as you battle to the finish line.

Make sure that at any point in time, you know the smallest unit you're working on; rather than just the big target for your project.

Define done

You've got an idea of the big picture; you know what you're ultimately trying to build. And you know the particular sub-task you're working on at the moment.

Now, make sure that for whatever task you are working on, you know when to stop.

To do this, you have to define what 'done' is. You have to know what 'success' means. What the 'complete' software will look like.

> Make sure you define 'done'.

This is important. If you haven't determined when to stop, you'll keep working far past when you needed to. You'll be working harder and longer than you needed to. Or, you won't work hard enough – you'll not get everything done. (Not getting everything done sounds easier, doesn't it? But it's not... the half-done work will come back to bite you, and will make more work for you later down the line, whether that's bugs, rework, or an unstable product).

Don't start a piece of coding work until you know what success is. If you don't yet know, make your first task determining what 'done' is. Only then,

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

Hotel Room to Train Carriage

Jon Jagger shares some illuminating musings.

Paul has asked me to write something for CVu. I don't have a clear idea of what to write about so I thought instead I would just write about stuff I'm doing. Tonight I'm currently sat in a hotel room in Langley (Berks) after a thoroughly excellent day at a client's site. I've been teaching day one of my C Foundation course. John and Ed suggested some ways the slides, and the words I use to describe the slides, could be improved. So the first thing I've been doing is making some mods to the slides.

The first was a simple int to pointer conversion where I initialised an `int` pointer with the hex integer 0xBEEF. I chose 0xBEEF because it spells a word but didn't think about alignment. 0xBEEF is an odd integer number and the address of an integer is likely to be, at the very least, even.

The second was something I said. In the chapter about pointers I said that an invalid pointer doesn't have to be dereferenced for bad things to happen.

JON JAGGER

Jon is a UK-based software consultant. His passion is helping people improve their effectiveness in this collaborative game we call software development.

When It's Done, It's Done (continued)

get going. With the certainty of knowing where you're headed, you'll be able to work in a focused, directed manner. You'll be able to make informed choices, and to discount unnecessary things that might side-track or delay you.

> If you can't tell when it's done, then you shouldn't start it.

So how does this look in practice? How do you define 'done'? Your 'done' criteria need to be:

Clear

It must be unambiguous and specific. A list of all the features to be implemented, the APIs added or extended, or the specific faults to be fixed.

If, as you get into the task, you discover things that might affect the completion criteria (e.g. you discover more bugs that need fixing, or uncover unforeseen problems) then you must make sure that you reflect this in your 'done' criteria.

This criteria is usually directly traceable to some software requirements or a user story – if you have them. If this is the case, make sure that this connection is documented.

Visible

Make sure that the success criteria is seen by all important parties. This probably includes: your manager, your customers, the downstream teams using your code, or the testers who will validate your work.

Make sure everyone knows and agrees on this criteria. And make sure they'll have a way of telling – and agreeing – when you are 'done'.

The nature of each task will clearly define what 'done' means. However you should consider:

- How much code must be completed. (Do you measure this in units of functionality, APIs implemented, user stories completed?)
- How much is design done, and how it's captured.
- Whether any documents or reports must be generated.

When it's a coding task, you can mostly clearly demonstrate 'being done' by creating an unambiguous test set. Write tests that will show when you've fashioned the full suite of code required.

> Use tests written in code to define when your code is complete and working.

There are some other questions that you may have to consider when you describe what 'done' is:

- Where is the code delivered to? (e.g. to version control)
- Where is the code deployed to? (Is it 'done' when it's live on a server – or do you deliver testable product ready for a deployment team to roll out?)
- What are the economics of 'done'? The exact numbers required that may lead to certain tradeoffs or measurements. For example: how well should your solution scale? It's not good enough if your software only manages 10 simultaneous users if 10,000 are required. The more precise your done criteria the better you understand these economics.
- How will you signal that you're done? When you think you're done how will you let the customer/manager/QA department know? This probably looks different for each person. How will you garner agreement that you are indeed done – who signs-off on your work? Do you just check in, do you change a project reporting ticket, or do you raise an invoice?

Just do it

When you've defined 'done', you can work with focus. Work up to the 'done' point. Don't do more than necessary.

Stop when your code is good enough – not necessarily perfect (there may be a real difference between the two states). If the code gets used or worked on an awful lot, it may eventually be refactored to be perfect – but don't polish it yet. This may just be wasted effort. (Beware: this is not an excuse to write bad code, just a warning against unnecessary over-polishing).

> Don't do more work than necessarily. Work until you're 'done'. Then stop.

Having a single, specific goal in mind helps you to focus on a single task. Without this focus it's easy to hack at code randomly trying to achieve a number of things and not managing any of them successfully. ■

Questions

1. Do you know when you're current task will be 'done'? What does 'done' look like?
2. Have you decomposed your current task into a single goal, or a series of simple goals?
3. Do you decompose your work into achievable, measurable units?

That's too strong. You only need to think about the `free()` function to realise that. After you've called `free(ptr)` then `ptr` no longer points to an object but as long as you don't dereference `ptr` you'll be ok. What I should have said is that when you do pointer arithmetic any sub-expressions must always be valid.

Something I try to do regularly is to examine what happened during my day and to learn from it. A sort of retrospective. So I'm looking at the C course mods and wondering what I can learn from them. I'm wondering, in the first example, why I chose 0xBEEF in the first place. Choosing a hex number that spells an English word is overly clever. Overly cute. So now I'm thinking I'd be better off choosing a hex number that doesn't remotely even look like a word. One that would align correctly. I'm struck by how hard it is to avoid being overly clever. By how hard it is to recognise and take into account context. From the second example, I'm reminded how hard it is to say clearly what I'm thinking. Or rather, how difficult it is to have sufficient clarity in my thinking that the words are naturally clear.

As well as learning from the day's mistakes it's also important to think about what went well. I recently did a '60 minutes in the brain of' presentation for SkillsMatter [1] London on deliberate practice (the topic of one of my entries in Kevin's *97 Things Every Programmer Should Know* [2]). SkillsMatter videoed the talk and I've watched it a couple of times. Watching yourself on video is a tremendously valuable thing. It allows you to see yourself how others see you. Not how you see yourself. For example, something I noticed was a tendency I have to say 'ok?' rhetorically at the end of a sentence. Ok? Doing that once or twice is fine but I did it perhaps a dozen times. It was almost becoming a tick. So one of my tasks today was to try not to say 'ok?' at the end of my sentences. It's hard to know if I achieved that I at least feel I didn't say it so often it became a noticeable tick. And I recall a couple of times where I was conscious I had not ended the sentence with 'ok?' when previously I might have. But, interestingly, as I write this, I'm thinking how can I know? Perhaps I'm still doing it but don't realise because I can't see myself. So I'm wondering if, tomorrow, I will explicitly ask the guys on the course to tell me if I do it. I think I will. As I write this I'm reminded of the line from Robert Burns's *To a Louse*, 'To see ourselves as others see us!' I remember Jerry Weinberg mentioning the connection in one of his books. And I notice bagpipe music is playing on my iTunes as I wrote that. I wonder if my subconscious made the Scottish connection? It's the *Skye Boat Song*. My mother used to sing that to me as a bedtime lullaby when I was a small boy.

It's now the next day. I'm on a train heading home. I'm reading some of *Weinberg on Writing* by Jerry Weinberg. I always carry a book. Usually three. I try to read a lot. I remember Craig Larman once saying to me once that one of the main things that marks a consultant as a consultant is they read a lot. I have an odd reading habit. I don't read one book from start to finish before starting another book. Instead I read a small chunk from one book, and then put it down, and read another small chunk from another book. I often have about 20 books all partly read. Today the other two are *A Little Book of f-Laws* by Russell Ackoff and Herbert Addison, and *Culture Against Man* by Jules Henry. I don't have any fixed rules about switching from one book to another. I just try to sense when it feels right. And I don't have any fixed rules about which books I pop into my travel bags either. It may sound strange but I am trying to cultivate by subconscious so again I just try to sense what feels right. As I'm reading I highlight text that speaks to. For example, in Jerry's book I've highlighted the following:

A trigger is a small amount of input energy that sets off a large amount of output energy.

That caught my interest. I recall Jerry discussing triggers in some of his other books. So I've made a note to reread what he said about triggers there in the light of that quote. An actual note that is – I also always carry a pen and paper. Here's another snippet:

Raise your typing speed by ten words per minute. This will give you an extra six hundred words for every hour you work. If you work an hour a day, two hundred days a year, you'll type an extra 120,000 words – a couple of books' worth.

That's quite illuminating. Here's another:

You know, there would be no problem raising kids if only you could throw away the first one.

That made me laugh. And one last one:

Writer's block is not a disorder in you, the writer. It's a deficiency in your writing methods – the mythology you've swallowed about how works get written – what my friend and sometime co-author Tom Gilb called your 'mythology'.

It took me a while to find that last one. I knew I had read it but I couldn't find it. I searched in the chapter I read most recently. No luck. I flicked through looking at the marked passages. No luck. Then I stopped and thought about what to do next. (It's almost always better than repeating a failed approach.) I wondered if Tom Gilb had an entry in the index. And he did. And it was exactly the passage I remembered. It was on page 19, not anywhere near the chapter I was reading most recently. And it was a passage I'd marked as I read it. And yet I looked at the marked passages trying to find it. I clearly missed it. That's reminded me that I have a tendency to only see something when searching for it if matches my pre-formed idea of what it will look like. Natalie (my wife) tells me that's a typical male trait. That last snippet about Tom Gilb has sparked something in my mind. Partly I'm attracted to the word myth because one of my favourite corny jokes is the old chestnut 'what is a myth?' the answer being 'a female moth'. But it's more than that. I think it's because my subconscious is connecting the word mythology with the word methodology. That's an interesting connection. A Mythodology perhaps? When I've finished reading a book I copy the bits I highlighted into my personal wiki. Then I copy the dozen or so highlights that speak to me the loudest into a small book 'review' snippet for my blog site *Less Code, More Software*. Then I put the book onto a pile to take to the next accu conference to raise some money for the charity.

We'll be coming into Taunton at moment now so I'm going to sign off. ■

References

[1] http://skillsmatter.com/

[2] http://programmer.97things.oreilly.com/wiki/index.php/97_Things_Every_Programmer_Should_Know

A Game of Tug o' War

Baron Muncharris sets a challenge.

Salutations Sir R-----! Pray come join me at my table and take a glass of this most palatable brandy.

Will you again join me in a little gaming with your refreshment?

Excellent!

I have in mind a game of dice inspired by that ancient sport of tug o' war; a sport of which I have some small experience.

I recall one particular contest upon the cloud girdled summit of Mount Olympus; the residents being especially fond of such diversions, as any fellow schooled in the classics will attest. I had been invited to attend at the behest of the noble Pallas Athene who wished to learn of the many exotic peoples and lands I had encountered during my travels.

As I was describing the extraordinary habits of those settlers of the New World an announcement came that the entire home team had been forced to withdraw from the event on account of a lack of funds, their having to a man entered into a series of ill-advised business ventures.

Having no desire to see my hosts thusly humiliated before the eyes of their neighbours I naturally offered to represent them in their stead. Madame Athene accepted my offer and I made my way to the field of sport, only to find that I was to be pitted against my own regiment of Hussars!

I took this as a very great stroke of luck as I had been away from the regiment for some time and might use this opportunity to remind my brothers in arms of the quality of my mettle.

Upon the instruction of the referee we first took up the strain and then began to heave. To my very great surprise these fellows were a match for me; the flag didn't move one inch! For an hour we struggled and strained with no discernable motion on either side.

With my patience wearing thin I resolved to give it my very all and, with one final tremendous heave, I got the better of my comrades. But as the flag passed over my line, the stand before me gave a tortured groan and collapsed; much to the consternation of those unfortunate souls perched upon it.

Upon inspection it was found that one of the supporting beams was no longer in its rightful place, but rather was tied to the end of the rope. Clearly I need not have worried that my reputation had faded at regimental head quarters!

But here I am delaying our sport!

See I have set before you a track of 12 squares.

0	1	2	3	4	5	5	4	3	2	1	0

The leftmost 6 squares, from the home square 0 up to the first 5, shall be mine and the rightmost 6 squares, from the second 5 down to the second home square 0, shall be yours.

To begin I shall cast a die and place a coin upon that square of mine so numbered. If I roll a 6 I shall pass the die to you and you shall do the same, but placing the coin upon the so numbered square of yours. In the unlikely event that you too throw a 6 we shall console ourselves with another draft of this splendid brandy and, upon having so refreshed ourselves, recommence the contest.

At each turn he whose square the coin rests upon shall again cast the die. If its value exceeds that upon the square he shall move it one square towards his home, otherwise he shall move it one square away.

Whosoever first moves the coin onto his home square shall have won the wager. If it is I then I shall have from you 3 coins whereas if it is you then you shall have from me 9 and one quarter part of a coin.

The reaction of that godforsaken student acquaintance of mine to the rules of this game was to start mumbling about his need to rehearse some symphony or other. I confess that I was glad of an excuse for keeping the exchange as brief as possible, but must admit my surprise at this unexpected musical propensity, it being hard to imagine that he is in possession of a talent of any kind! Presumably he passes his evenings playing low-brow tunes for measures of genever at those disreputable establishments one supposes he so often frequents.

But enough of this! Here, take another draft and mull over your want for a wager!

Listing 1 shows a C++ implementation of the game. ■

Listing 1

```
unsigned
roll()
{
  return 1 + unsigned(6.0 * double(rand()) /
                      (double(RAND_MAX)+1.0));
}

bool
play()
{
  bool barons_move = true;
  unsigned square = roll();

  while(square==6)
  {
    barons_move = !barons_move;
    square = roll();
  }

  while(square!=0)
  {
    const unsigned heave = roll();

    if(heave>square)   --square;
    else if(square!=5) ++square;
    else barons_move = !barons_move;
  }

  return !barons_move;
}
```

BARON MUNCHARRIS

In the service of the Russian military Baron Muncharris has travelled widely in this world, and many others for that matter, defending the honour and the interests of the Empress of Russia. He is renowned for his bravery, his scrupulous honesty and his fondness for a wager.

On a Game of Roulette

A student analyses the Baron's last puzzle.

You will recall that the Baron proposed two games at a roulette wheel, both for a stake of one coin. In the first Sir R----- was to spin the wheel three times and mark the point closest to him after each spin. If the triangle thus formed enclosed the centre of the wheel he should have received a prize of four coins. In the second he should have won six and one half coins if a point chosen at random upon the face of the wheel was inside said triangle.

When figuring Sir R-----'s expected winnings in these games, we shall make lighter work of it if we recognise the puzzles' symmetry under rotation; after Sir R----- has chosen his three points we can spin the wheel again to reveal another, equally likely, outcome. When I mentioned this to the Baron it seemed to me that it upset him a little, although I can fathom no reason as to why it should have done so.

We can further exploit symmetry under reflection; looking at the wheel in a mirror also reveals an equally likely outcome.

Now, in the first game, these observations mean that we can assume that the first point is always at the top of the wheel, at twelve of the clock as it were, since we are free to rotate the wheel until this is so. We can further assume that the second point lies to its right, clockwise between the top and the bottom of the wheel since we can view the wheel through a mirror if it does not. The third point will, after these manipulations, still lie upon some random spot upon the edge of the wheel.

If we draw a line from the second point through the centre of the wheel we shall discover a fourth point upon its edge. It is plain to see that the third point must lie clockwise between the bottom of the wheel and this fourth point if the triangle is to enclose the centre of the wheel.

Figure 1

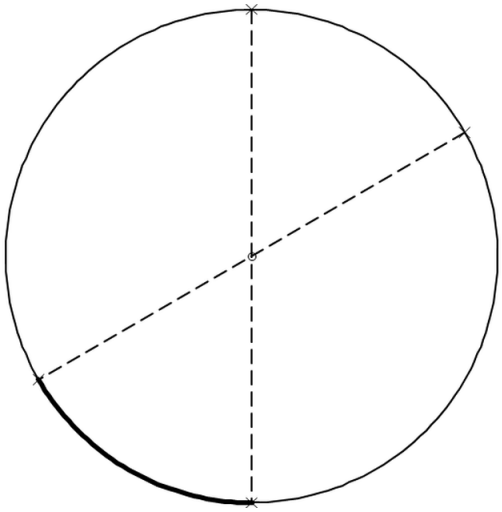


Figure 1 shows the set of winning third points marked in bold.

If the angle between the lines from the centre of the wheel to the first and second points is θ, then the probability that the triangle will enclose the centre is equal to

$$\frac{\theta}{2\pi}$$

The expected winnings of the game are equal to the average of this probability over all such θ multiplied by the prize of four coins, given by

$$\frac{4}{\pi} \times \int_0^{\pi} \frac{\theta}{2\pi} d\theta = \frac{4}{\pi} \times \left[\frac{\theta^2}{4\pi}\right]_0^{\pi} = 1$$

Since this is equal to Sir R-----'s stake I should have advised him that it was a fair game and he should have no compunction in playing if he so wished.

The winnings Sir R----- might expect from Baron's second game are a little more difficult to reckon. If we were to try the same trick that we used for the first game we should soon find ourselves tied up in knots.

We might try figuring the outcome of a number of games using pencil and paper, but if we do so we had better take care that the points on the face of the wheel are chosen with uniform probability; that the probability of a chosen point lying within a given region is some constant multiple of the area of that region.

When I explained this to the Baron his temper worsened considerably and he turned on his heel and left. I must confess that I remain utterly ignorant of how I might have offended him!

Now, it is not sufficient to pick a random angle and distance from the centre of the wheel when choosing points since this will concentrate the points at the centre of the wheel. A superior scheme by far is to pick points at random from within a square that surrounds the wheel and ignore those that do not lie upon its face (Listing 1).

In playing some few hundred games on paper, my fellow students and I found that the game seemed fair, but we were by no means certain.

Then it dawned upon me that, since the point is picked uniformly upon the face of the wheel, the probability of winning the game must be equal to the average area of the triangles divided by the area of the wheel.

To figure the average area of the triangles we can once again exploit the symmetries of the game. Specifically we rotate the wheel so that the first and second points lay either side of the line joining the top and bottom of the wheel and at the same height from the bottom. We may now assume that the third point will lie to the right of this line, since we can reflect the wheel if it does not (Figure 2).

To simplify matters, we shall use the radius of the wheel as our unit of length.

Now the area of such a triangle is equal to half of the length of the base multiplied by the height of the third point above, or depth below, it.

If the angle between the lines connecting the centre of the wheel and the top and rightmost point on the base is θ then, with a little trigonometry, we find that the length of the base is equal to

$$b = 2\sin\theta$$

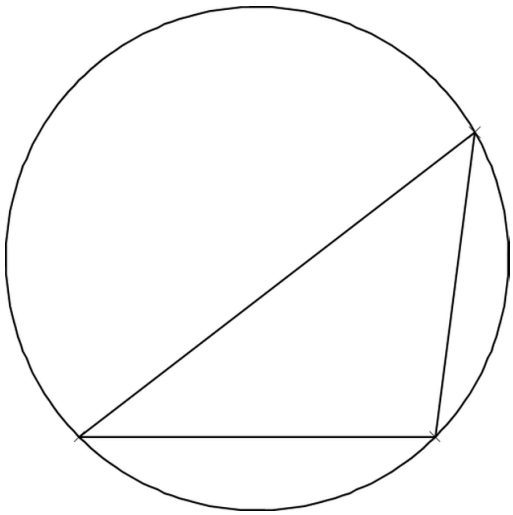
Listing 1

```
point
pick()
{
  double x, y;

  do
  {
    x = 2.0*double(rand())/
        (1.0+double(RAND_MAX)) - 1.0;
    y = 2.0*double(rand())/
        (1.0+double(RAND_MAX)) - 1.0;
  }
  while(x*x+y*y>=1.0);

  return point(x, y);
}
```

Figure 2



Similarly, if the angle so formed with the third point is α, then the height of the triangle is given by

$$h = \cos\alpha - \cos\theta$$

giving an area of

$$A = \sin\theta\cos\alpha - \sin\theta\cos\theta$$

If α is greater than θ then this area will be negative so we shall have to take care that we do not carelessly subtract the areas of such triangles from the average.

We shall do so by breaking the calculation of the average area into two parts. Firstly those triangles whose third point is above the base line and secondly those triangles whose third point is below it.

For a given θ, the average area of the triangles is thusly given by

$$\frac{1}{\pi}\times\left[\int_0^{\theta}(\sin\theta\cos\alpha - \sin\theta\cos\theta)da + \int_0^{\pi}-(\sin\theta\cos\alpha - \sin\theta\cos\theta)da\right]$$

$$= \frac{1}{\pi}\times[\sin\theta\sin\alpha - \alpha\sin\theta\cos\theta]_0^{\theta} - \frac{1}{\pi}\times[\sin\theta\sin\alpha - \alpha\sin\theta\cos\theta]_0^{\pi}$$

$$= \frac{\sin^2\theta - \theta\sin\theta\cos\theta}{\pi} - \frac{-\sin^2\theta - (\pi-\theta)\sin\theta\cos\theta}{\pi}$$

$$= 2\frac{\sin^2\theta - \theta\sin\theta\cos\theta}{\pi} + \sin\theta\cos\theta$$

To calculate the average area of any triangle we must perform a similar exercise upon this result.

$$\frac{2}{\pi^2}\int_0^{\pi}\sin\theta d\theta - \frac{2}{\pi^2}\int_0^{\pi}\sin\theta\cos\theta d\theta + \frac{1}{\pi}\int_0^{\pi}\sin\theta\cos\theta d\theta$$

The third term is the simplest to figure since

$$\frac{d}{d\theta}\sin^2\theta = 2\sin\theta\cos\theta$$

Since sin θ is equal to zero when θ is equal to both zero and π, this term is simply zero.

To figure the second term, we must use a technique known as integration by parts which states

$$\int u\frac{dv}{dx}dx = [uv] - \int v\frac{du}{dx}dx$$

If we take u to be θ, this yields

$$\int_0^{\pi}\theta\sin\theta\cos\theta d\theta = \frac{1}{2}[\theta\sin^2\theta]_0^{\pi} - \frac{1}{2}\int_0^{\pi}\sin^2\theta d\theta$$

The first of these terms is also zero, so the average area of the triangles must equal

$$\frac{3}{\pi^2}\int_0^{\pi}\sin^2\theta d\theta$$

Using integration by parts again, with both u and v equal to sin θ, we have

$$\int_0^{\pi}\sin^2\theta d\theta = [-\sin\theta\cos\theta]_0^{\pi} - \int_0^{\pi}-\cos^2\theta d\theta = \int_0^{\pi}\cos^2\theta d\theta$$

Adding the integral of the squared sine to both sides of this equation yields

$$2\int_0^{\pi}\sin^2\theta d\theta = \int_0^{\pi}\cos^2\theta d\theta + \int_0^{\pi}\sin^2\theta d\theta = \int_0^{\pi}(\cos^2\theta + \sin^2\theta)d\theta$$

and hence

$$\int_0^{\pi}\sin^2\theta d\theta = \frac{1}{2}\int_0^{\pi}(\cos^2\theta + \sin^2\theta)d\theta = \frac{1}{2}\int_0^{\pi}1 d\theta = \frac{\pi}{2}$$

The average area of the triangles is therefore

$$\frac{3}{\pi^2}\times\frac{\pi}{2} = \frac{3}{2\pi}$$

Dividing this by the area of the wheel, which is trivially equal to ? in the units we have adopted, and multiplying by the prize yields the expected winnings of this game

$$\frac{3}{2\pi}\times\frac{1}{\pi}\times 6\frac{1}{2} = \frac{39}{4\pi^2} < \frac{99}{100}$$

The game is consequently slightly biased in the Baron's favour and I could not in good conscience have advised Sir R----- to play.

Whilst my fellow students and I were considering the Baron's games we came to wonder what might make a fair prize if the point were chosen on the face of the wheel *before* the wager was made.

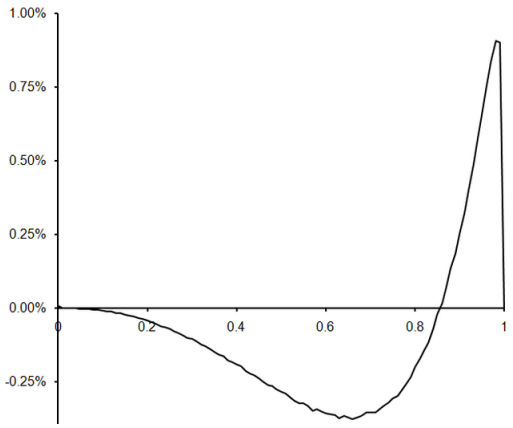
Considering the symmetry under rotation, it was readily apparent to us that the size of such bounty should depend only upon the distance between the point and the centre of the wheel.

By way of a careful approximation I found that the probability of winning such a wager was always within roughly one chance in a hundred of

$$\frac{\arccos r + r(1-r)}{2\pi}$$

where arccos is the inverse function of the cosine and r is the distance of the point from the centre of the wheel using the radius of the wheel as the unit of length. Unfortunately not one of us has managed to bring this puzzle to a tidy conclusion. I hardly need add that the fact that so simply stated a wager has entirely evaded our most strenuous efforts has been a source of no little frustration to me and my fellows. Figure 3 shows the error in the approximation against r. ■

Figure 3



Somno, The Barber of Clapham Junction, Introduces GPars

Russell Winder introduces concurrency techniques in Groovy.

The Java Virtual Machine (JVM) has been around since before Java became popular – remember the programming language and virtual machine were called Oak as part of Project Green, before it all got rebranded Java in 1995. From the outset it has supported threads. Initially this meant user space threads in a uniprocessor process, but as operating systems started to support kernel threads, JVM threads migrate to being kernel threads. As operating systems harnessed kernel threads for managing multiple processors, this meant that the JVM got parallelism for free: as long as the operating system scheduled kernel threads across all processors available then the JVM threads could potentially execute in parallel.

The Multicore Revolution over the last three or four years has terminated the era of the ever increasing clock speed of uniprocessors, and replaced it with the ever increasing count of cores on a processor chip. The era of the dual core processors is long gone, quad core is now the norm, 12-core is on the horizon, and 48-core will soon be going into production – at least for server chips, if not workstation and laptop chips.

Now whilst C++0x may (or may not) soon be ratified, this just brings a standard thread model, albeit with good things such as futures and asynchronous function calls. But this is low level infrastructure. Moreover, it still assumes shared-memory multithreading, and hence the need for locks, semaphores, monitors, and the ability to program all this concurrency technology. Go [1] and D [2] have both chosen to use lightweight processes and message passing as the way forward for managing concurrency and parallelism. Is this process and message passing technology something there should be C++ support for? Almost certainly yes if C++ is to remain relevant in the coming times. C++, Go and D though are in native code territory, what about the JVM?

The main thrust of Java development has been to accept that explicit shared-memory multithreading is not the right approach for application programming in a multi-processor context: the facilities in java.util.concurrent have become the proper tools for handling parallelism in Java code. This replaces explicit thread management with the use of 'executors' – abstracted ways of working with thread and process pools – along with futures, asynchronous function calls, and parallel arrays. Underneath it remains shared-memory multithreading requiring locks, semaphores, monitors, atomics, etc., but the programmer works with a façade, with a few extra facilities, to make it more usable and less prone to error.

Scala [3] a language that, like Java, targets the JVM, took the direction of realizing Actor Model as the principle tool of concurrent and parallel programming. People can still use threads and shared-memory multithreading if they really, really have to, but they are considered as low-level tools for realizing a higher level model of computation (actors) more suitable for day-to-day programming of concurrent and parallel systems.

The Actor Model was formulated in the early 1970s, but seems to have been studiously ignored by the mainstream programming languages for reasons that only the language designers involved will ever be able to shed light on. Likely though, they probably won't be telling – as they probably don't know themselves.

JCSP [4] has been around since 1997 and has been keeping the candle burning for process-based computation in Java since before Scala was around (c.2003): JCSP is a realization of Communicating Sequential Processes (CSP, [5]) for Java. Sadly, most Java programmers have never heard of CSP, let alone JCSP.

CSP was initially expressed by Tony Hoare around 1978, but it only really hit the headlines in 1984 when his book (*Communicating Sequential Processes*, [6]) was published. As with Actor Model, the mainstream computer languages studiously ignored this model, again for reasons which are unlikely to come to light. Conspiracy theories possibly lead to prejudice as the reason – CSP was perceived as mathematics not programming.

Another process-based model is 'Dataflow Model'. Many people associate 'dataflow' as a term with either:

- 'dataflow diagrams' and the software analysis and design techniques popularized by Tom DeMarco [7], and Ed Yourdon and Larry Constantine [8]; or
- the attempt to create 'dataflow computers', i.e. hardware.

Whilst there were some sterling efforts to create dataflow computers, dataflow never really took off as a computational model for hardware. However, the abstract architecture makes an excellent software architecture. The underlying 'computational model' in both these variants of 'dataflow' is in fact fundamentally the same, data flowing along channels between operations. The fact that it works for software whereas it probably doesn't in hardware is a definite opportunity.

The models

The Actor Model, CSP, and Dataflow Model rely on using processes and message passing. Processes are distinct namespaces and/or address spaces. This does not necessarily mean separate processors, processes can be realized by partitioning a global address space – Erlang has been using this approach very successfully for many years. This approach is obviously very appropriate for today's multicore processors with multiple processors sharing a single memory. Processes can of course be separate address spaces on separate processor. This leads to an interesting issue: where does multicore end and distributed begin? This is really a question of communications. There are many levels of communications speed:

1. Bus on chip.
2. Bus on motherboard.
3. Local board cluster.
4. Local machine cluster.
5. Wide area network.

One of the difficulties of the moment for these process-based models is that they have just a single notion of communication: it is assumed that all processes are at just one of the communications levels. Knowing the cost of communication between processes is something that is critically important to the programmer of an algorithm. It will therefore soon have to be the case that weights will have to be given to the communications of the various processes so as to take into account where a target process is

RUSSEL WINDER

Having done the Professor of Computing Science and Head of Department thing, I tried the CTO thing. Just as things were going well the accountants closed the company. I am now doing the author, trainer, consultant thing – all reasonable jobs considered.

relative to the sending process. Parallel and distributed computing will have to merge into a single area of study.

Caveat communications cost, however any of these three models is realized, as far as the programmer is concerned, each process is completely independent and can only pass messages to other processes. There is (or, at least, should not be) any notion of shared state. Clearly though in a single memory realization of processes it can be tempting to utilize shared memory, such temptation must be fought against and eschewed.

So the commonality is that the three models are all process based. The differences between them lies in the way in which messages are passed and the synchronization behaviour that is integral to the processing of messages.

The actor model

Processes in the Actor Model each have one, and only one, incoming message queue, often called a mailbox. An actor can send a message to any other actor for which it has a reference to the target actor's message queue. How a sending actor gets a reference to the message queue of the receiving actor is not predetermined, it could be by being given a reference during construction or a reference might be sent as part of a message.

Messages are sent asynchronously in the Actor Model: the sender adds a message to the message queue of the receiver and continues execution. Each actor is responsible for processing messages in its queue. In effect an actor is an event processing system with the message queue being the event list.

CSP

CSP uses the idea of processes being connected by channels. Channels can be one-to-one, one-to-many, many-toone, or many-to-many. Each process has zero or more input channels and zero or more output channels. As with Actor Model, there is an element of being event driven: a process is responsible for taking messages from its input channels, doing computation, and putting messages out on its output channels.

Apart from the fact that each process can have many rather than one input message queue, the other crucial difference between actors and CSP is that message passing is synchronous in CSP: message passing is actually a rendezvous between two processes. This makes it relatively straightforward to create systems that deadlock, but if it happens, it is surprisingly easy to discover what the deadlock is and how to fix it. Also because the processes are sequential and the channel properties can be modelled exactly with mathematics, it makes it (relatively) easy to write systems that reflect on code and determine whether or not that code will deadlock. The beauty of CSP is that you can find out with certainty whether your code will exhibit deadlock or not. If the tool for determining deadlock says your code can't deadlock, then it won't, this is not something that can be done for shared memory multithreading or Actor Model.

Dataflow model

In the Dataflow Model the processes are usually referred to as operators – a hang over from the days of dataflow computer hardware most likely. Like CSP processes, Dataflow Model operators can have zero or more inputs and zero or more outputs. Unlike CSP and like Actor Model, Dataflow Model has asynchronous message sending. However whereas Actor Model and CSP require an explicit activity in the process for receiving messages, Dataflow Model pins the execution and synchronization behaviour of an operator to the reception of messages on its inputs. The simplest of the algorithms is for the operator to be idle until there is valid input on all the incoming channels. Having valid inputs on all channels is the event that triggers execution in the operator, which then does something and puts data out on the output channels. Alternative algorithms are possible, for example trigger execution on reception of the first message on any input.

Which model when

It may not be obvious at this stage but it is possible to implement any of the three process-based models in any of the others. However to do so would likely lead to inefficient systems. It is far better to implement all three independently using the low-level thread and process management of the underlying platform. This means treating the three as distinct. The different message receive and execution trigger properties of the models become the factors for determining what facilities of what packages to use for a given solution to a given problem: some solutions to some problems will more naturally be solved using one model rather than another. So what is a Java programmer to do? java.util.concurrent has some great tools but...Groovy and GPars provide massively useful augmentations.

GPars

GPars is a Groovy attempt to bring Actor Model, CSP, and Dataflow Model to the Java and Groovy communities – Scala people can also use it, but this seems less likely to happen for various reasons, some technical, most psychological and/or social. The idea of using Groovy as a platform for creating a framework for managing concurrency and parallelism on the JVM had been mooted a number of times on the Groovy mailing lists in 2008 and early 2009. Václav Pech decided to act rather than talk, and created the GParallelizer project. This was a personal project to get things moving. Mid-2009 various people (including myself) joined Václav on the project. After a little debate, we moved the project to Codehaus [9], the home of Groovy [10], and rebranded it GPars [11]. Since then Václav has been tireless in moving things forward supported to a greater or lesser extent by the other members of the team.

Why use Groovy rather than just Java, or Scala? Well the Scala concurrency and parallelism support is ploughing its own furrow based on Actor Model, and supporting frameworks such as Akka and Scalaz. The Groovy effort, driven via GPars, is about providing high-level, low-overhead augmentation of what is available in Java. GPars harnesses the meta-object protocol of Groovy to provide a very easy way of expressing concurrent and parallel computations.

Of course all this is very general and vague, what is needed is something less abstract, something more concrete. Basically we need some examples.

Concurrency (and parallelism), the classic problems

Since multiprogramming was first invented, operating systems programming has been spawning problems in concurrency and share memory management. The 1960s and 1970s saw an entire industry in educating people how to manage resources using semaphores, monitors and locks. Sadly these problems, and techniques for solution, were foisted on applications programmers as well as systems programmers, but that rant is wholly, but not completely, inappropriate for this article.

The classic problem in concurrency and resource management that every programmer is forced to study is the 'Dining Philosophers Problem'. The underlying issue is one of resources being managed by operating systems – and the origins of 'Dining Philosophers' is just such a problem set out by Edsgar Dijkstra, but Tony Hoare reformulated it in real-world terms so as to make it more comprehensible. Thus began two industries: a) providing multiple solutions inmultiple languages to the 'Dining Philosophers Problem'; and b) finding new and novel model problems for concurrency problems in operating systems couched as real-world vignettes.

The 'Sleeping Barber Problem' (also attributed to Edsgar Dijkstra) is just such a problem. Like 'Dining Philosophers Problem' it is a model of a process synchronization issue in operating systems. But as Tony Hoare showed, these things are far more interesting, and take on much more life, when couched in real-world terms. Enter Somno, The Barber of Clapham Junction.

Despite rumours, it is known that Somno does not have brother called Figaro living in Seville. Moreover Somno does not sing, not even in a barber's shop quartet.

The sleeping barber problem

Somno owns a barber shop. He has one hair-cutting chair and four waiting chairs. Music is provided via Somno's MP3 player, an amplifier and speakers, there is no barber's shop quartet, Somno does not sing. If there are no customers Somno snoozes in the hair-cutting chair. If there are any customers, Somno cuts hair of customers, one at a time. Potential customers enter the shop and if they see Somno asleep in the chair, they wake him up, take the seat themselves and get a hair cut. If Somno is cutting another customer's hair then the potential new customer checks the four waiting chairs to see if one is vacant. If there is a free chair the customer sits and waits their turn. If there are no free chairs, the person exits – somewhat less than chuffed at not being able to transform from potential customer to customer. Potential customers arrive at random intervals, and each customer hair cut takes a random amount of time.

At some point Somno is thinking of getting a second, or even a third hair-cutting chair, allowing for there to be more than one barber in action concurrently. For now though cash flow doesn't allow for this to be implemented. Somno is saving up for a new pair of scissors.

Using threads

The Wikipedia article [12] describes things much more according to the operating system problem that inspired the description above. The customers are processes or threads as is the barber: the problem is all about synchronizing and queuing processes or threads. The article presents a pseudocode solution based on using semaphores. This can be coded up very easily in Java or Groovy using threads and the `java.util.concurrent.Sempahore` class, so as to avoid either writing an implementation of semaphores or using the Java wait/notify technology explicitly – which generally results in incomprehensibly deadlocked code. Here is a Groovy version, the Java version would be fundamentally the same, just more verbose (see Listing 1).

A few thoughts and points on this code:

1 Groovy realizes the `#!` magic number start word specified by Posix for executables. This means Groovy scripts can be commands on Posix-compliant systems. Those addicted to Windows use will just have to remain green with envy. Either that or join the ranks of those using a proper operating system.

36 Groovy allows for list literals, e.g. `[]`, the resultant object is of type `ArrayList` by default.

58 Groovy allows for overloading of operators – like C++ and completely unlike Java. The `<<` in this context is appending a value to a list.

62 The operator `*.` here is a 'spread apply' operation. The left-hand operand is a list and the right-hand operand is the method call to be applied to each element in the list.

68 Groovy has the facility to create anonymous functions that can be passed as parameters: code in curly brackets in a context other that one of 'code block' create a function object. Groovy gives these things the type Closure which is actually a bit of a misnomer, they are not closures (a closure is a function with an environment providing bindings for all the free variables in the function) they are functions potentially with free variables. In this case it happens not to be the case, so we can't tell that anonymous functions are not actually closures. Which is good.

There are undoubtedly other interesting, or at least quite interesting, points about the code, but let us leave it there for now. If there are questions that are not answered by the end of the article, feel free to email me, or perhaps raise the issue on the ACCU General email list.

Running the above program will result in something such as shown in Listing 2. Since there is an element or two of randomness in the code, no two runs of the code are guaranteed to be the same.

Which is (quite) interesting, not least because Somno appears to start cutting the hair of the first customer before that customer has arrived in the shop!

Listing 1

```
#! /usr/bin/env groovy

// This is a model of the "The Sleeping Barber" problem using Groovy (http://groovy.codehaus.org),
// cf. http://en.wikipedia.org/wiki/Sleeping_barber_problem.
//
// Copyright (c) 2010 Russel Winder

import java.util.concurrent.Semaphore

def runSimulation ( final int numberOfCustomers , final int numberOfWaitingSeats ,
                    final Closure hairTrimTime , final Closure nextCustomerWaitTime ) {
  final customerSemaphore = new Semaphore ( 1 )
  final barberSemaphore = new Semaphore ( 1 )
  final accessSeatsSemaphore = new Semaphore ( 1 )
  def customersTurnedAway = 0
  def customersTrimmed = 0
  def numberOfFreeSeats = numberOfWaitingSeats
  final barber = new Runnable ( ) {
    private working = true
    public void stopWork ( ) { working = false }
    @Override public void run ( ) {
      while ( working ) {
        customerSemaphore.acquire ( )
        accessSeatsSemaphore.acquire ( )
        ++numberOfFreeSeats
        barberSemaphore.release ( )
        accessSeatsSemaphore.release ( )
        println ( 'Barber : Starting Customer.' )
        Thread.sleep ( hairTrimTime ( ) )
        println ( 'Barber : Finished Customer.')
      }
    }
  }
```

Listing 1 (cont'd)

```
34   final barberThread = new Thread ( barber )
35   barberThread.start ( )
36   final customerThreads = [ ]
37   for ( number in 0 ..< numberOfCustomers ) {
38     println ( "World : Customer ${number} enters the shop." )
39     final customerThread = new Thread ( new Runnable ( ) {
40                                         public void run ( ) {
41                                           accessSeatsSemaphore.acquire ( )
42                                           if ( numberOfFreeSeats > 0 ) {
43                                             println ( "Shop : Customer ${number} takes a seat.
                                                ${numberOfWaitingSeats - numberOfFreeSeats} in use." )
44                                             --numberOfFreeSeats
45                                             customerSemaphore.release ( )
46                                             accessSeatsSemaphore.release ( )
47                                             barberSemaphore.acquire ( )
48                                             println ( "Shop : Customer ${number} leaving trimmed." )
49                                             ++customersTrimmed
50                                           }
51                                           else {
52                                             accessSeatsSemaphore.release ( )
53                                             println ( "Shop : Customer ${number} turned away." )
54                                             ++customersTurnedAway
55                                           }
56                                         }
57                                       } )
58     customerThreads << customerThread
59     customerThread.start ( )
60     Thread.sleep ( nextCustomerWaitTime ( ) )
61   }
62   customerThreads*.join ( )
63   barber.stopWork ( )
64   barberThread.join ( )
65   println ( "\nTrimmed ${customersTrimmed} and turned away ${customersTurnedAway} today." )
66 }
67
68 runSimulation ( 20 , 4 , { ( Math.random ( ) * 60 + 10 ) as int },
69    { ( Math.random ( ) * 20 + 10 ) as int } )
```

One of the problems with the code above is that is it trying to (minimally) solve the thread/process scheduling problem, annotated with pointers to the analogy, rather than being a model of Somno's barber shop. There are therefore some inconsistencies between the activity and the analogy used to describe the problem. The solution is to forget the operating system problem that inspired the analogy and to model and realize the analogy more directly, i.e. take a much more simulation-oriented view of the whole problem.

Also of course, the above is really a lesson in how not to do things unless you are writing an operating system – and who would want to write an operating system in Groovy, or even Java.

There are rumours that a number of universities are using Java as the programming language for exercises in operating systems courses – bizarre.

So what we should do is move much more towards a simulation modelling approach. The core change is that customers become represented as data rather than being labels on processes. This models far more literally what happens in Somno's shop. (Listing 3)

Note that we have to allow Somno to finish all the waiting customers (cf. 48, 23, 49) before actually shutting up shop, even after it is closed. It would be somewhat uncivilized to allow customers to seat themselves expecting a hair cut only to be ushered out of the door, sans trim, just because Somno wants to knock off and go home.

An alternative might be for Somno to start trying to sing when he wants to go home. The appalling noise would almost certainly clear the shop in a huge hurry.

Now whilst the above is a (relatively) simple, two-thread solution with the waiting chairs being the thread-safe shared state, there is a lot of data

Listing 2

```
World : Customer enters shop.
Barber : Starting Customer.
Shop : Customer takes a seat. -1 in use.
Shop : Customer leaving trimmed.
World : Customer enters shop.
Shop : Customer takes a seat. 0 in use.
Shop : Customer leaving trimmed.
World : Customer enters shop.
Shop : Customer takes a seat. 1 in use.
Barber : Finished Customer.
Barber : Starting Customer.
Shop : Customer leaving trimmed.
World : Customer enters shop.
Shop : Customer takes a seat. 1 in use.
Barber : Finished Customer.
Barber : Starting Customer.
Shop : Customer leaving trimmed.
World : Customer enters shop.
Shop : Customer takes a seat. 1 in use.
World : Customer enters shop.
Shop : Customer takes a seat. 2 in use.
World : Customer enters shop.
Shop : Customer takes a seat. 3 in use.
Barber : Finished Customer.
Barber : Starting Customer.
Shop : Customer leaving trimmed.
World : Customer enters shop.
Shop : Customer takes a seat. 3 in use.
World : Customer enters shop.
```

Listing 2 (cont'd)

```
Shop : Customer turned away.
Barber : Finished Customer.
Barber : Starting Customer.
Shop : Customer leaving trimmed.
World : Customer enters shop.
Shop : Customer takes a seat. 3 in use.
World : Customer enters shop.
Shop : Customer turned away.
Barber : Finished Customer.
Barber : Starting Customer.
Shop : Customer leaving trimmed.
World : Customer enters shop.
Shop : Customer takes a seat. 3 in use.
World : Customer enters shop.
Shop : Customer turned away.
World : Customer enters shop.
Shop : Customer turned away.
Barber : Finished Customer.
Barber : Starting Customer.
Shop : Customer leaving trimmed.
World : Customer enters shop.
Shop : Customer takes a seat. 3 in use.
World : Customer enters shop.
Shop : Customer turned away.
World : Customer enters shop.
Shop : Customer turned away.
Barber : Finished Customer.
Barber : Starting Customer.
Shop : Customer leaving trimmed.
World : Customer enters shop.
Shop : Customer takes a seat. 3 in use.
World : Customer enters shop.
Shop : Customer turned away.
World : Customer enters shop.
Shop : Customer turned away.
Barber : Finished Customer.
Barber : Starting Customer.
Shop : Customer leaving trimmed.
Barber : Finished Customer.
Barber : Starting Customer.
Shop : Customer leaving trimmed.
Barber : Finished Customer.
Barber : Starting Customer.
Shop : Customer leaving trimmed.
Barber : Finished Customer.

Trimmed 12 and turned away 8 today.
```

coupling. For example notification about customers leaving the shop are handled in the barber thread. Just because Somno has finished cutting the customers hair doesn't imply the customers has actually paid up and left the shop. Also the world and shop are being modelled with the same thread. In effect we haven't modelled separately the idea of world and shop, which would properly decouple things.

So perhaps we should write a new version with a barber object, a shop object, perhaps even a world object, each having a Singleton instance, and animated with many threads.

Mentioning Singleton here is of course offering the proverbial red rag to the proverbial bull, especially as the readership here are ACCU members. I have no doubt there will be a re-emergence of the traditional anti-Singleton thread on ACCU General email list if anyone actually reads this bit of this article.

An analytic intervention

Is it obvious? We are rapidly moving towards creating a number of objects (world, shop, barber) each animated with a thread with each object communicating to other objects by passing customers around. This sounds like processes and message passing à la Actor Model, CSP, or Dataflow Model!

It should be no surprise to anyone that trying to enforce encapsulation and decoupling in this simulation modelling approach leads, especially in an object-oriented programming context, to processes and message passing. After all, object-orientation is all about processes and message passing: the object-oriented model resulted from investigating simulation, cf. Simula-67.

Using shared-memorymultithreading explicitly at the same time as claiming to be following an object-oriented approach seems to be an act of blatant doublethink.

Let's short-circuit showing the whole sequence of solutions between above and below and jump straight to GParsbased solutions – thereby avoiding reimplementing GPars, which is effectively what would happen.

Actor model

This section presents a version of the Sleeping Barber Problem in which each entity is modelled with an actor. There are actors for barber, shop and world: well at least barber and shop, the world is modelled using the initial thread that executed the program.

Before reading the following code, it is probably worth noting that the **@Grab** annotation is one specific to Groovy (it is part of the Grapes system in Groovy) that uses Ivy under the covers to ensure that the named artifact is in the classpath for execution of the code, downloading the artifact if necessary. In the case below the **Grab** is ensuring that the GPars artifact

Listing 3

```
#! /usr/bin/env groovy

// This is a model of the "The Sleeping Barber" problem using Groovy (http://groovy.codehaus.org)
// only, cf. http://en.wikipedia.org/wiki/Sleeping_barber_problem.
//
// Copyright (c) 2010 Russel Winder

import java.util.concurrent.ArrayBlockingQueue

import groovy.transform.Immutable

@Immutable class Customer { Integer id }

def runSimulation ( final int numberOfCustomers , final int numberOfWaitingSeats ,
                    final Closure hairTrimTime , final Closure nextCustomerWaitTime ) {
  final waitingChairs = new ArrayBlockingQueue<Customer> ( numberOfWaitingSeats )
  final customersTurnedAway = 0
  final customersTrimmed = 0
  final barber = new Runnable ( ) {
```

Listing 3 (cont'd)

```
20      private working = true
21      public void stopWork ( ) { working = false }
22      @Override public void run ( ) {
23        while ( working || ( waitingChairs.size ( ) > 0 ) ) {
24          def customer = waitingChairs.take ( )
25          assert customer instanceof Customer
26          println ( "Barber : Starting Customer ${customer.id}." )
27          Thread.sleep ( hairTrimTime ( ) )
28          println ( "Barber : Finished Customer ${customer.id}." )
29          ++customersTrimmed
30          println ( "Shop : Customer ${customer.id} leaving trimmed." )
31        }
32      }
33    }
34    final barberThread = new Thread ( barber )
35    barberThread.start ( )
36    for ( number in 0 ..< numberOfCustomers ) {
37      Thread.sleep ( nextCustomerWaitTime ( ) )
38      println ( "World : Customer ${number} enters the shop." )
39      final customer = new Customer ( number )
40      if ( waitingChairs.offer ( customer ) ) {
41        println ( "Shop : Customer ${customer.id} takes a seat. ${waitingChairs.size ( )} in use." )
42      }
43      else {
44        ++customersTurnedAway
45        println ( "Shop : Customer ${customer.id} turned away." )
46      }
47    }
48    barber.stopWork ( )
49    barberThread.join ( )
50    println ( "\nTrimmed ${customersTrimmed} and turned away ${customersTurnedAway} today." )
51  }
52
53  runSimulation ( 20 , 4 , { ( Math.random ( ) * 60 + 10 ) as int },
54     { ( Math.random ( ) * 20 + 10 ) as int } )
```

version 0.11 is used which is the latest version of GPars in the Maven repository.

*Grapes has to be deemed **extremely** cool.*

(See Listing 4)

Various notes about the code:

15 Message passing in the Actor Model invariably requires the use of case classes. Since an actor has but a single message queue, the type of a message becomes very important metadata that drives the computation. The shop has to process messages coming from both the world and the barber and there has to be a way of distinguishing which message came from where. The barber therefore (25) returns

Listing 4

```
 1  #! /usr/bin/env groovy
 2
 3  // This is a model of the "The Sleeping Barber" problem using Groovy (http://groovy.codehaus.org)
 4  // and GPars (http://gpars.codehaus.org) actors, cf. http://en.wikipedia.org/wiki/
 5  // Sleeping_barber_problem.
 6  // Copyright (c) 2009-10 Russel Winder
 7
 8  @Grab ( 'org.codehaus.gpars:gpars:0.11' )
 9
10  import groovy.transform.Immutable
11
12  import groovyx.gpars.group.DefaultPGroup
13
14  @Immutable class Customer { Integer id }
15  @Immutable class SuccessfulCustomer { Customer customer }
16
17  def runSimulation ( final int numberOfCustomers , final int numberOfWaitingSeats ,
18                      final Closure hairTrimTime , final Closure nextCustomerWaitTime ) {
19    def group = new DefaultPGroup ( )
20    def barber = group.reactor { customer ->
21      assert customer instanceof Customer
22      println ( "Barber : Starting Customer ${customer.id}." )
23      Thread.sleep ( hairTrimTime ( ) )
```

Listing 4 (cont'd)

```
24      println ( "Barber : Finished Customer ${customer.id}." )
25      new SuccessfulCustomer ( customer )
26    }
27    def shop = group.actor {
28      def seatsTaken = 0
29      def isOpen = true
30      def customersTurnedAway = 0
31      def customersTrimmed = 0
32      loop {
33        react { message ->
34          switch ( message ) {
35            case Customer :
36              if ( seatsTaken <= numberOfWaitingSeats ) {
37                ++seatsTaken
38                println ( "Shop : Customer ${message.id} takes a seat. ${seatsTaken} in use." )
39                barber.send ( message )
40              }
41              else {
42                println ( "Shop : Customer ${message.id} turned away." )
43                ++customersTurnedAway
44              }
45              break
46            case SuccessfulCustomer :
47              --seatsTaken
48              ++customersTrimmed
49              println ( "Shop : Customer ${message.customer.id} leaving trimmed." )
50              if ( ! isOpen && ( seatsTaken == 0 ) ) {
51                println ( "\nTrimmed ${customersTrimmed} and turned away
                     ${customersTurnedAway} today." )
52                stop ( )
53              }
54              break
55            case '' : isOpen = false ; break
56            default : throw new RuntimeException ("Shop got a message of unexpected type
                 ${message.class}" )
57          }
58        }
59      }
60    }
61    for ( number in 0 ..< numberOfCustomers ) {
62    Thread.sleep ( nextCustomerWaitTime ( ) )
63    println ( "World : Customer ${number} enters the shop." )
64    shop.send ( new Customer ( number ) )
65  }
66  shop.send ( '' )
67  shop.join ( )
68 }
69
70 runSimulation ( 20 , 4 , { ( Math.random ( ) * 60 + 10 ) as int },
71                          { ( Math.random ( ) * 20 + 10 ) as int } )
```

a customer wrapped as a **SuccessfulCustomer** so that the shop can make the distinction. cf. 34–57.

19 Thread pools are used to animate the actors, hence the notion of a group: all actors in a given group will be animated by the threads in the associated thread pool. In this case we have just the one group.

20 Somno is realized as a reactor. A reactor is an actor that has no persistent state, it simply processes received message and returns a message to the sender of the received message. A nice abstraction for Somno the barber. Somno sleeps if there are no customers (modelled by being blocked waiting for an item on the message queue: the message queue models the waiting chairs in the shop) and reacts to a new customer by giving them a hair cut and sending them back to the shop wrapped as a **SuccessfulCustomer**

61–67 The world is not an actor just the initial thread sending in customers to the shop. Note that we use an empty string as a marker to close the shop. Messages can be of any type, the receiving actor must deal with this. so in the switch statement (34–57) we have 55 which deals with this signal to close the shop, sent in 66.

The complexity in the shop actor is due to the need to managing persistent state. It is an actor, but not a simple reactor, so therefore it needs a react loop. So there is a **loop** construct (32–59) and a **react** construct (33–58), which defines the function to execute on reception of a message. loop is just a function that takes a 'closure' parameter and executes that 'closure' according to the loop constraints – in this case loop infinitely. react is a function that takes a one-parameter 'closure' that gets called for each message received by the actor. The shop receives customers from the outside world and from the barber. Hence the need for the **switch** and **case** classes in the **react** function to distinguish where the customer has come from. The shop keeps track of how many customers there are in Somno's waiting queue, and is responsible for generating statistics for the day's business.

Note that we use the sending in of an empty string instead of a customer instance as the marker to close the shop. As previously though we must

let the barber process all the waiting customers before it is home time. This way of closing the shop – processing a fixed number of customers and then sending in a special message – is clearly not a good 'physical' simulation, but we'll live with it for now.

Groovy CSP

Groovy CSP is a Groovy layer over JCSP – all the power of CSP as implemented in JCSP, with all the ease and simplicity of Groovy as a language for creating internal, domain specific languages. Sadly at the time of writing Groovy CSP hasn't had the work done that it deserves, so the code is a little bit ugly. The hope and intention is to evolve the Groovy CSP system to make much of what is in fact boilerplate code, expressible in a neater and shorter way. (Listing 5)

Some commentary:

21 No extra 'case class'. Because CSP allows for multiple input channels, the shop can distinguish messages from the world and from the barber by having them to be sent on different channels. If we had a many-to-one channel for both the world and the barber to send

Listing 5

```
#! /usr/bin/env groovy

// This is a model of the "The Sleeping Barber" problem using Groovy (http://groovy.codehaus.org)
// and Groovy CSP (a part of GPars, http://gpars.codehaus.org), cf. http://en.wikipedia.org/wiki/
// Sleeping_barber_problem.
// Copyright (c) 2010 Russel Winder

@Grab ( 'org.codehaus.jcsp:jcsp:1.1-rc5' )
@Grab ( 'org.codehaus.gpars:gpars:0.11' )

import groovy.transform.Immutable

import org.jcsp.util.Buffer
import org.jcsp.lang.Channel
import org.jcsp.lang.CSProcess

import groovyx.gpars.csp.PAR
import groovyx.gpars.csp.ALT

@Immutable class Customer { Integer id }

def runSimulation ( final int numberOfCustomers , final int numberOfWaitingSeats ,
                    final Closure hairTrimTime , final Closure nextCustomerWaitTime ) {
  final worldToShopChannel = Channel.one2one ( )
  final shopToBarberChannel = Channel.one2one ( new Buffer ( numberOfWaitingSeats ) )
  final barberToShopChannel = Channel.one2one ( )
  final barber = new CSProcess ( ) {
    @Override public void run ( ) {
      final fromShopChannel = shopToBarberChannel.in ( )
      final toShopChannel = barberToShopChannel.out ( )
      while ( true ) {
        final customer = fromShopChannel.read ( )
        if ( customer == '' ) { break }
        assert customer instanceof Customer
        println ( "Barber : Starting Customer ${customer.id}." )
        Thread.sleep ( hairTrimTime ( ) )
        println ( "Barber : Finished Customer ${customer.id}." )
        toShopChannel.write ( customer )
      }
    }
  }
  final shop = new CSProcess ( ) {
    @Override public void run ( ) {
      final fromBarberChannel = barberToShopChannel.in ( )
      final fromWorldChannel = worldToShopChannel.in ( )
      final toBarberChannel = shopToBarberChannel.out ( )
      final selector = new ALT ( [ fromBarberChannel , fromWorldChannel ] )
      def seatsTaken = 0
      def customersTurnedAway = 0
      def customersTrimmed = 0
      def isOpen = true
    mainloop:
      while ( true ) {
        switch ( selector.select ( ) ) {
          case 0 : //////// From the Barber ////////
            def customer = fromBarberChannel.read ( )
```

Listing 5 (cont'd)

```
57              assert customer instanceof Customer
58              --seatsTaken
59              ++customersTrimmed
60              println ( "Shop : Customer ${customer.id} leaving trimmed." )
61              if ( ! isOpen && ( seatsTaken == 0 ) ) {
62                println ( "\nTrimmed ${customersTrimmed} and turned away
                     ${customersTurnedAway} today." )
63                toBarberChannel.write ( '' )
64                break mainloop
65              }
66              break
67            case 1 : //////// From the World ////////
68              def customer = fromWorldChannel.read ( )
69              if ( customer == '' ) { isOpen = false }
70              else {
71                assert customer instanceof Customer
72                if ( seatsTaken < numberOfWaitingSeats ) {
73                  ++seatsTaken
74                  println ( "Shop : Customer ${customer.id} takes a seat. ${seatsTaken} in use." )
75                  toBarberChannel.write ( customer )
76                }
77                else {
78                  println ( "Shop : Customer ${customer.id} turned away." )
79                  ++customersTurnedAway
80                }
81              }
82              break
83            default :
84              throw new RuntimeException ( 'Shop : Selected a non-existent channel.' )
85          }
86        }
87      }
88    }
89    final world = new CSProcess ( ) {
90      @Override public void run ( ) {
91        def toShopChannel = worldToShopChannel.out ( )
92        for ( number in 0 ..< numberOfCustomers ) {
93          Thread.sleep ( nextCustomerWaitTime ( ) )
94          println ( "World : Customer ${number} enters the shop." )
95          toShopChannel.write ( new Customer ( number ) )
96        }
97        toShopChannel.write ( '' )
98      }
99    }
100   new PAR ( [ barber , shop , world ] ).run ( )
101 }
102
103 runSimulation ( 20 , 4 , { ( Math.random ( ) * 60 + 10 ) as int },
104                          { ( Math.random ( ) * 20 + 10 ) as int } )
```

messages to the shop, then we would need case classes – but why bother when we can just use multiple one-to-one channels.

24–26 We must explicitly create the channels. The channel for the world to communicate to the shop and the channel for the barber to communicate to the shop are both simple one-to-one channels that require rendezvous between the processes. the channel for the shop to send messages to the barber has a buffer. It is this that introduces the element of asynchronous behaviour in the communication between shop and barber. This just models the waiting seats of course.

27, 42, 89 Barber, shop and world are processes: the way CSP works everything has to be a process, we cannot 'get away with it', as we did in the actor version in the previous section, using the initial thread to represent the world.

100 Once all the processes are set up, and all the channels connected to the processes, we have to create a process that determines how the other processes execute. In this case we start all the processes in parallel and let the communication of messages between the processes determine what happens.

47, 54–85 Because message passing is synchronous in CSP and the shop has multiple input channels, it has to have a way of choosing between the channels. So in 48 we create an 'alting' object. The select method of this object blocks until there is a message available on one of the channels and return the index in the original list of the channel that is ready to read. There are therefore two 'branches' of control flow, one for a barber channel message and one for a world channel message.

33, 51, 61–65, 69, 97 As with the Actor Model version in the previous section, an 'out of band' message, i.e. a message with an unusual type that constitutes a case class, is used to signal termination. The world sends an empty string message to the shop just prior to terminating. The shop uses this empty string message to set the shop state to closed. When the barber has cut the last customer's hair, the shop sends an empty string message to the barber telling it to terminate, and then terminates itself.

52, 64 Crikey, a label, and what effectively amounts to a goto. Should Somno declare 'shock, horror, . . . , probe'? Not really. The problem here is that break on its own is relevant to the switch and we need to have a break out of the while. The option of introducing a Boolean to handle this seems overcomplicated compared to using the 'break out of a labelled statement' feature of Java and Groovy.

There are of course many other ways of structuring this code. For example we could have explicitly defined classes for barber, shop and world with constructors. Somno and I will leave this as an 'exercise for the student'.

Dataflow model

The following solution to The Sleeping Barber Problem not only uses Dataflow Model, it uses a lot of 'short cut' features that GPars provides. Structurally the code is not dissimilar to the Groovy CSP version in the previous section. This is entirely reasonable: the algorithm is fundamentally similar except that we are using asynchronous submission of values to dataflow queues instead of synchronous communication via CSP channels. Another aspect of the the reason this code looks cleaner and simpler than the Groovy CSP code is that more work has already gone into the 'dataflow DSL' compared to Groovy CSP.

Some commentary:

24 Attempting to obtain the next value from the `DataFlowQueue` (remember Groovy has properties, in this case `shopToBarber.val` means `shopToBarber.getVal ( )`) causes a block pending availability of a value – Somno sleeps until there is a customer of whom to cut the hair.

Listing 6

```
#! /usr/bin/env groovy

// This is a model of the "The Sleeping Barber" problem using Groovy (http://groovy.codehaus.org)
// and GPars (http://gpars.codehaus.org) dataflow, cf. http://en.wikipedia.org/wiki/
// Sleeping_barber_problem.
// Copyright (c) 2010 Russel Winder

@Grab ( 'org.codehaus.gpars:gpars:0.11' )

import groovy.transform.Immutable

import groovyx.gpars.dataflow.DataFlow
import groovyx.gpars.dataflow.DataFlowQueue

@Immutable class Customer { Integer id }

def runSimulation ( final int numberOfCustomers , final int numberOfWaitingSeats ,
                    final Closure hairTrimTime , final Closure nextCustomerWaitTime ) {
  def worldToShop = new DataFlowQueue ( )
  def shopToBarber = new DataFlowQueue ( )
  def barberToShop = new DataFlowQueue ( )
  final barber = DataFlow.task {
    while ( true ) {
      def customer = shopToBarber.val
      if ( customer == '' ) { break }
      assert customer instanceof Customer
      println ( "Barber : Starting Customer ${customer.id}." )
      Thread.sleep ( hairTrimTime ( ) )
      println ( "Barber : Finished Customer ${customer.id}." )
      barberToShop << customer
    }
  }
  final shop = DataFlow.task {
    def seatsTaken = 0
    def customersTurnedAway = 0
    def customersTrimmed = 0
    def isOpen = true
  mainloop:
    while ( true ) {
      def selector = DataFlow.select ( barberToShop , worldToShop )
      def item = selector.select ( )
      switch ( item.index ) {
        case 0 : //////// From the Barber ////////
          assert item.value instanceof Customer
          --seatsTaken
          ++customersTrimmed
          println ( "Shop : Customer ${item.value.id} leaving trimmed." )
          if ( ! isOpen && ( seatsTaken == 0 ) ) {
            println (
               "\nTrimmed ${customersTrimmed} and turned away ${customersTurnedAway} today." )
            shopToBarber << ''
            break mainloop
          }
          break
```

Listing 6 (cont'd)

```
54          case 1 : //////// From the World ////////
55            if ( item.value == '' ) { isOpen = false }
56            else {
57              assert item.value instanceof Customer
58              if ( seatsTaken < numberOfWaitingSeats ) {
59                ++seatsTaken
60                println ( "Shop : Customer ${item.value.id} takes a seat. ${seatsTaken} in use." )
61                shopToBarber << item.value
62              }
63              else {
64                println ( "Shop : Customer ${item.value.id} turned away." )
65                ++customersTurnedAway
66              }
67            }
68            break
69          default :
70            throw new RuntimeException ( 'Shop : Selected an non-existent queue.' )
71        }
72      }
73    }
74    for ( number in 0 ..< numberOfCustomers ) {
75      Thread.sleep ( nextCustomerWaitTime ( ) )
76      println ( "World : Customer ${number} enters the shop." )
77      worldToShop << new Customer ( number )
78    }
79    worldToShop << ''
80    // Make sure all computation is over before terminating.
81    [ barber , shop ]*.join ( )
82  }
83
84  runSimulation ( 20 , 4 , { ( Math.random ( ) * 60 + 10 ) as int },
85                           { ( Math.random ( ) * 20 + 10 ) as int } )
```

30, 50, 61, 77, 79 The **<<** operator targeting a **DataFlowQueue** is an operator overload and means insert the item in the queue. Probably obvious to C++ people, less so to Java people.

25, 50, 55, 79 An empty string message remains the tool for signalling the end of computation.

40–41 The shop has two input queues. Rather than waiting for valid input on both, we need to ensure we process valid values as soon as they are available. Hence a selector on which we can select. This is directly analogous to the select mechanism in CSP, except that we are dealing with queues and not channels. In GPars, the select method returns an object that packages the available value and the index of the selector from which the value was retrieved.

As always there are many, many variants that could be written using the same underlying infrastructure. In this case (Dataflow Model), we could use explicit operators. Somno (and I) are of the opinion that this is an excellent 'exercise for the student'. Of course it may be that we could be convinced to write a follow up article presenting these alternate solutions.

Reflections

Ignoring the first version, and looking at the other four – threads, actors, CSP, dataflow – it is clear that the threads version has fewer lines of code. In this case, I suggest that more is more. Although the threads version is shorter, I claim the other three are actually easier to comprehend. Of course the trouble is I wrote them. I guess I need to not read the code for six months and then see.

Despite the similarities between the actors, CSP and dataflow versions, a different mental model of execution is required – the different message passing semantics have to be handled explicitly.

A very CSP oriented viewpoint has been used to write the dataflow version, and so the dataflow version looks more like the CSP version than perhaps it should – at least in trying to present actors, CSP and dataflow as three separately useful tools.

Anecdotal experience indicates that the actor, CSP and dataflow versions are much easier not to get wrong compared to the threads version.

Summary

Of course these solution are unlikely to be useful as solution to the operating system resource management problem per se, not least because it is highly unlikely that anyone will ever write an operating system in a mix of Groovy and Java. There is though the distinct possibility that D and/or Go will eventually be used to write an operating system. Since D implements Actor Model and Go implements CSP (more or less), some of the ideas in this article may eventually become part of operating system orthodoxy.

It might even be the case that C++ gets Actor Model and Dataflow Model infrastructure based on the new C++0x standard. There is already a CSP implementation, C++CSP2, but it is not C++0x, it is just C++99.

Who said dynamic languages had no place in performance code – despite being extraordinarily slow in comparison to Java, huge tracts of applications are not performance critical, and Groovy is easily fast enough for those bits. ■

Acknowledgements

Thanks to Václav Pech and Paul King, fellow committers in the Groovy and GPars projects for various conversations and code examples that didn't make it explicitly into this article but helped shape what did get in. Also they gave splendidly valuable feedback on a draft of this article.

References

[1] http://go-lang.org
[2] http://d-programming-language.org
[3] http://www.scala-lang.org/
[4] http://www.cs.kent.ac.uk/projects/ofa/jcsp/
[5] http://www.usingcsp.com/

A Foray into CMake

Colin Hersom tells us of his experience using CMake for the first time.

CMake (Cross Platform Make) is described as 'a family of tools designed to build, test and package software' [1].

I am only concerned here in its use as a platform-independent Makefile generator. As such it is an alternative to the GNU autotools (autoconf, automake & libtool) [2]. These are venerable products, used by many open source projects for as long as I remember and were explored by Jez Higgins a few years ago [3].

I hope to give you a quick sketch of my experience of CMake, without dwelling too much on the minutiae of the process. I am running Ubuntu 10.4 [4]. Other operating systems may behave differently.

Broken-down auto

A few months ago I picked up an abandoned project (untouched for three years). Over time its environment (KDE [5]) had changed and some of the programs that it was relying on had been replaced. A small fix was required to make it find newer versions.

This sounds simple enough, but as always there are unexpected obstacles and sometimes the best way of reaching ones destination is not to remove the obstacle, but to find a completely different route. Although I did not know it at the time, this was going to be one of those occasions.

After configuring in a separate build directory and running **`make`**, the compiler complained of missing header files. It was looking in a directory relative to the build because the `Makefile.am` file (and hence the final Makefile) had a line of the form:

```
INCLUDES          = -I../core $(all_includes)
```

I tried fixing this, but for some reason my automake was missing some macros. It seems obvious that this project had never been compiled in a separate build directory. Maybe this is the reason it lies abandoned. The original author has vanished and no-one can work out how to build it in its current state. What do I want to do? Well the GUI is KDE3 and yet the world has moved on to KDE4. If I want to make this project up-to-date, I am going to need to convert to KDE4. One of the many changes that KDE4 has introduced is that it uses CMake in place of autotools, so rather than fighting automake, a switch to CMake now would seem to make sense.

Let's see what I am to make

Since many projects needed to move from autotools to CMake in their transition to KDE3, a simple means of porting is available. This is in the form of the program **`am2cmake`**. Run in the top source directory, this recursively examines the `Makefile.am` files and creates the corresponding `CMakeLists.txt` files. The results indicate that this is a very crude conversion. For this project, which has a number of directories each building a shared-object and one building an executable, the top-level `CMakeLists.txt` pulls in KDE4 and QT and some other stuff then drops into the subdirectories. In the subdirectories, the source files are found and an appropriate target (either library or executable) is defined. It does not find the dependencies on other packages (since configure did this) nor, surprisingly, does it find the header files that need to be processed using QT's automoc. Although I have to add it manually, using that relative path for include files actually works now, since all such paths are resolved relative to the source directory

The expectation of the use of KDE4 is possibly not surprising, but is a little annoying in this case. I need to replace this with KDE3 and alter the macro calls like **`kde4_add_library`** with the base function **`add_library`**. Ensuring that required packages are available is easy in CMake since you use the macro **`find_package`**. For a specified package, this searches

COLIN HERSOM

Colin Hersom was born into computers and has been programming all his adult life. He is currently between assignments and is looking at some open source projects to keep himself amused.

Somno, The Barber of Clapham Junction, Introduces GPars (continued)

[6] Hoare, Tony (1984) *Communicating Sequential Processes*, Prentice-Hall, 1985

[7] DeMarco, Tom (1979) *Structured Analysis and System Specification*, Prentice-Hall

[8] Yourdon, Ed and Constantine, Larry (1979) *Structured Design:Fundamentals of a Discipline of Computer Program and System Design*, Prentice-Hall

[9] Codehaus, http://www.codehaus.org

[10] Groovy, http://groovy.codehaus.org

[11] GPars, http://gpars.codehaus.org

[12] *Wikipedia*, Sleeping Barber Problem, http://en.wikipedia.org/wiki/Sleeping_barber_problem

Postscript

The code in this article, and many other not disimilar variants in many other languages, can be found in browseable form at http://www.russel.org.uk:8080/SleepingBarber. This is a Bazaar branch being rendered by Loggerhead. If you want to branch the branch using Bazaar then use the url http://www.russel.org.uk/Bazaar/SleepingBarber.

Endnote

If you want to get involved in testing or developing GPars, just get stuck in and/or ask on the mailing lists. If you want to do things surreptitiously, then just clone the GPars Git repository which is at git://git.codehaus.org/gpars.git and research from there. The browsable version is at http://git.codehaus.org/gitweb.cgi?p=gpars.git.

cmake's path for a file called `Find<package>.cmake` and this tests to see whether the appropriate files are on the system. Since the file name is made by concatenating strings, the package name is case sensitive, in contrast to most of the rest of CMake. It is not always obvious what the package is called, for example, for CUPS there is `FindCups.cmake` but for curl it is `FindCURL.cmake`. The set of packages supported by the CMake included with KDE3 is considerably smaller than with KDE4, yet these packages are not dependent on KDE at all. In order to build a KDE3 program, I needed to copy some of the 'Find' files from the KDE4 CMake directories into my local directory.

Tsunami warning

Having pulled in all the required packages, it was time to try compiling again. This resulted in a deluge of warning messages from the compiler, so many that error messages at the beginning of a compilation were impossible to find simply by watching the terminal window. A redirection of the output like this:

```
make 2 > error.txt
```

worked well since the progress messages appeared on the screen but all the errors went to the file.

Why were there suddenly so many warnings? The flags given to the compiler are in a file called flags.make which is constructed by CMake and put into `<dir>/CMakeFiles/<dir>.dir`, where `<dir>` is the subdirectory that we are building in. This shows that every warning that could be thought of is being produced. Although I agree with this, it is unfortunate that QT3 triggers quite so many of them!

Closed library

Now I haven't given my progress in strict chronological order. This project consists of a number of components which produce non-UI shared objects and a GUI executable that is dependent on KDE3. I can thus build the shared objects without care to the version of KDE installed. I can use the ready-compiled executable which comes in the Debian package to test whether my new libraries are compatible. So I had done this with KDE4 macros. Trying to run this showed that the libraries were missing all linker symbols. Investigating the flags.make file again shows that the compiler is being given these flags:

```
-fvisibility=hidden -fvisibility-inlines-hidden
```

This conceals all functions and classes unless appropriate attributes are assigned to those that need to be exported. The reasoning behind this is sound, GCC exports every non-local symbol which makes it difficult to control the interface to a shared library. It does, however, effectively make the library useless until such attributes are defined. I expected macros (e.g. **`DLLEXPORT`**) to be defined to implement these attributes, but I could not see any. Until CMake provides a mechanism to tell the compiler which symbols to export, there is a problem since I cannot expect the code to be portable unless I switch those flags off. The KDE3 version does not add these flags.

Mockery

I mentioned earlier that the files to be processed through QT's automoc were not found with **`am2cmake`**, despite the `Makefile.am` containing the lines

```
# let automoc handle all of the meta source files
 # (moc)
 METASOURCES = AUTO
```

Listing 1

```
foreach(f_name ${gui_SOURCES})
  string(REGEX REPLACE "\\.cpp" ".h" HEADER ${f_name})
  set(gui_HEADERS ${gui_HEADERS} ${HEADER})
endforeach(f_name)
KDE3_ADD_MOC_FILES(gui_SOURCES ${gui_HEADERS} )
```

In order to get the build to work, this process (which adds additional routines for QT) needs to be run. The simplest way, assuming one header per C++ file, is to generate all the header file names from the source files and flag them to be run by automoc (Listing 1).

And now everything seems to work perfectly. I can add my small fix and rebuild which leaves me with a working program again.

A sea change?

My feelings about CMake are mixed. I am sure that I have only scratched the surface, but from this experience I feel that the positive points are:

- All the requirements are in one (set) of files, rather than distributed between autoconf & automake. The learning curve is thus less steep.
- Changing `CMakeLists.txt` and running make automatically calls CMake, but CMake caches most of its results, so only a small part of the configuration procedure is re-run. This is much faster that the autoconf equivalent.
- There is less (no?) danger of accidentally using relative paths.
- The installation directories can be chosen at install time, whereas this is fixed at configuration time with autotools'
- It runs on Windows as well as *n*x.

and the negatives:

- Some of its configuration files are tied in with the KDE version for no obvious reason.
- The 'is it or isn't it case sensitive?' question is a trap for the unwary. Having to search for the CMake 'Find' files to determine the correct case to use is a niggle I could do without.
- To compile a project, you must have CMake installed. This is a dependency that autotools do not have, since the 'configure' is self-contained'
- The lack of support for exporting library symbols is a problem.

When the tide has gone out

The autotools are mature and well used. CMake is the new kid on the block with great potential. The documentation is lacking, probably not surprising, but it does tend to mean delving into the source to find out how to achieve certain things. For the project that I am attempting to revitalise, it worked well, but I am sure that it is not to everyone's taste. For a new project, I would give it serious consideration, even if I were considering only using standard Makefiles, since CMake files are even easier to write. For an existing project with much more complex requirements, it is probably not worth the effort to change from autotools if that is working satisfactorily.

Now I'm off to wade through the debris of all those warning messages. ■

References

[1] www.cmake.org
[2] www.gnu.org/software
[3] CVu Dec 2006, Feb & Apr 2007
[4] www.ubuntu.org
[5] www.kde.org

Agile Cambridge 2010

Giovanni Asproni gives us an alternative view of Agile Cambridge 2010.

The first edition of the Agile Cambridge [1] conference took place this year on the 14th and 15th of October. I went there with great expectations, and I was not disappointed. The programme was quite interesting and the speakers included quite a few of the usual suspects: Rachel Davies, Nat Pryce, Steve Freeman, Allan Kelly, Jon Jagger, Paul Grenyer, and yours truly, while the audience was made mostly of people new to Agile.

I attended the two keynotes – one from James Whittaker, and the other from Rachel Davies – the talks from Allan Kelly, and Paul Grenyer, and, most importantly, the social event at the The Castle pub, where we had free food (which was surprisingly good) acompanied by free beer, courtesy of the conference sponsors.

James Whittaker is the Engineering Director over engineering tools and testing for Google's Seattle and Kirkland offices. He spoke about how they do testing at Google using the metaphor of hospital triages where testers are the doctors, and software applications the patients. Using this metaphor he described various tools they developed to be able to test software efficiently and to find and fix bugs quickly. He also described the 'tour' metaphor that he and his teams use to classify the kind of tests they run on a particular application. I found his keynote quite inspiring, and his book *Exploratory Software Testing* ended up very quickly in my Kindle.

Rachel spoke about building trust in agile teams. Setting aside lots of interesting material about the importance of trust in teams and on various techniques to use or avoid in order to earn trust, the highlight of her keynote was an exercise where Paul Grenyer was volunteered by Rachel to do a stage diving (interestingly enough, Rachel, Allan, Paul and I had talked about it the night before at the pub, but we didn't think Rachel was going to take the conversation seriously). He accepted and was caught by a group of six or eight people (which included Jon Jagger and Allan Kelly who joined them to make sure the ACCU didn't loose one of its most valued members). I'm happy to report that Paul was not hurt during the exercise (neither were Jon and Allan).

Paul presented a session entitled 'Agile is a journey not a destination' where he described his experience in introducing agile development practices at his company. The session was aimed at people trying to introduce agile in their own companies for the first time. Paul presented the material in a clear and compelling way, and, judging from the number of questions at the end, the audience really enjoyed it. Personally, I found the content quite interesting, and I was truly impressed by the way he delivered the presentation.

Allan's talk title 'What does it take to be an Agile company?' describes its content quite well. The session was very well led (as expected from Allan) and full of interesting points.

All in all a very interesting conference. I'm already looking forward to Agile Cambridge 2011. ■

Reference

[1] http://www.agilecambridge.net

Inspirational (P)articles

Dr Love introduces Sue Black.

At the age of seven Sue felt compelled to spend her pocket money on maths text books. She had very few friends. If only she had known then that one day geeks would be cool. Dr Sue Black is a Senior Research Associate in the Software Systems Engineering group at University College London and a Senior Consultant with Cornerstone Global Associates. She campaigns passionately to raise awareness and support for women in tech and Bletchley Park. Her research interests are software quality, software development paradigms, social media, public engagement and anything shiny. To find out more check out www.sueblack.co.uk and follow @Dr_Black on Twitter.

I found programming very hard during my degree, I think I'm quite a top down type of learner and we were taught programming bottom up. I just couldn't get a proper handle on what I was supposed to be doing and why, I found it exasperating. Despite this I managed to get a degree in computer science, not excelling in software engineering.

After my degree I started a PhD in formal methods, which I also wasn't particularly good at, but I loved research, I really wanted to do a PhD, and there was funding in that area. After six months I moved over to software measurement, felt comfortable and stayed there. Two years or more into my PhD I was designing and building a prototype tool in C. The tool was required to take C programs as input and calculate a ripple effect measure [1] for each module within a program and then the ripple effect for each program as a whole. I spent a lot of time angsting about how crap I was at programming, and then one day just somehow got on with it. It was hard to start with, but using my C books, the one I remember best being Kernighan and Ritchie [2] I gradually started to understand many things that I had just not understood before. The day that I coded a function from scratch to implement some complicated part of the ripple effect algorithm still stands in my memory. I wrote the code, I compiled it, had no syntax errors, I ran it and got what looked like reasonable output. I tested it and found that it gave the correct result. That was one of the best days of my life :)) ■

References

[1] Black, S. E. 'Computing ripple effect for software maintenance', *Journal of Software Maintenance and Evolution: Research and Practice 2001*; 13:263-279.

[2] Kernighan, Brian W.; Dennis M. Ritchie (February 1978). *The C Programming Language (1st ed.)*. Englewood Cliffs, NJ: Prentice Hall.ISBN 0-13-110163-3.

Desert Island Books

Rachel Davies shares her choice of books and music.

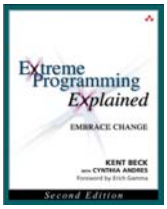
Like so many others I first encountered Rachel Davies at the ACCU Conference. I went to her session on effective story writing to discover it was a tutorial! I'm not a big fan of tutorials, but I enjoyed this one as it covered a lot of material I was reading about and trying to apply at the time. The second time I encountered Rachel was in a pub in Cambridge following the first day of the first Agile Cambridge conference. I somehow managed to agree to jump backwards off the stage the next day. And, sure enough as you can read elsewhere in this very CVu, I did it!

Rachel Davies

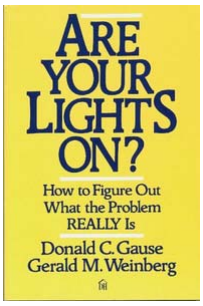
I've got around 20 years' experience as a software developer working mostly in C, C++, and Java. Nowadays, I'm not writing much software, instead I work as an independent coach helping teams work out how to apply agile practices effectively. My claim to fame is having written the first book on 'Agile Coaching' with Liz Sedley, which is published by Pragmatic Bookshelf. I'm also a bit of a conference addict and have been involved in organising XPDays, SPA and lots of other Agile events.

I live in Warwickshire with two teenage daughters and four cats. When I get a bit of spare time I like to work in my garden, grew potatoes and rhubarb for the first time this year. I also like getting out to do some hill walking and seeking out ancient standing stones. I've always loved books and always have a stack of books on software development that I'm working through. I've tried to pick the titles that have most influenced my approach to software development rather than what I'm reading right now.

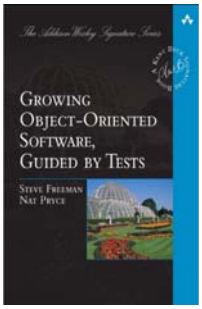
My first choice is *The Timeless Way of Building* by Christopher Alexander, which introduced me to patterns and the quest for the quality without a name and inspiration to trust in emergent design. Before reading this book, I tended to think of design as being an economical way of consolidating functionality to enable reuse without aesthetic value. Although Alexander writes about the design of buildings and towns, the photographs and drawings show how important it is to consider the people who live and work in the buildings. From this book, I grew to understand how important it is to strive to make our code habitable and pleasant to work in. Reading this book also lead me to the design patterns community and that's where I started to hear about refactoring and extreme programming (XP).



This brings me onto my next book choice. There's no question I have to pick *Extreme Programming Explained* by Kent Beck (although 1st edition, please). Reading this book, lead me to a totally new way of developing code working test-first and pair programming. This book gave me the courage to resign from my management job, go to eXtreme Tuesday Club and find a job as a programmer in an XP team. After several years of working on large C++ projects that always got cancelled for one reason or another, I was able to learn Java and finally deliver working software!



Now for a less obvious choice. The book is *Are your lights on? How to figure out what the problem REALLY is* by Donald C. Gause and Gerald M. Weinberg. This book is full of cartoons and stories that help the reader remember that they have some responsibility to figure out what problem they are solving. This book doesn't help you construct code but it can help you avoid writing unnecessary code that solves the wrong problem. I happen to value my time and so I think it's rather important to check my understanding of the requirements before getting stuck into writing code. It is always good to be reminded that 'The fish is always the last to see the water'.My last choice on software takes me back to Test-Driven Development (TDD). Although it was only published this year, *Growing Object-Oriented Software, Guided by Tests* by Steve Freeman and Nat Pryce explains a style of programming that I have been trying to apply since 2000. This was when I first started to learn how use of Mock Objects in unit tests shaped the design of the code making it more object oriented. It took a while to understand how important it is to work from the outside-in as the book emphasises. I've been lucky enough to work on a team with Steve and see first-hand how he listens to what the tests are telling us about the design. Nat and Steve have done a great job with this book explaining this approach to design with plenty of code examples and refelection about the alternative approaches that these reveal to design.



I typically only read non-fiction on airplanes or on holiday! Picking a novel is a tough choice, as I want to pick one that I can read many times while stuck on the desert island. So I'll go for *Wuthering Heights* by Emily Brontë. If you've seen film adaptations, you may have the impression that this is a basic love story. However, when you read the book then you discover that the full story is a complex tale of madness, isolation and revenge. It also has an interesting structure with shifting narrators and plenty atmospheric descriptions of the Yorkshire moors.

As for music, the album I listen to most is *Apollo: Atmospheres & Soundtracks* by Brian Eno. I play this when I'm writing as it helps me tune out distractions. I suppose peace and quiet is not a problem I'll have plenty of tranquility on the island. So instead, I'd like to take the *Hot Rats* album by Frank Zappa, which I've recently rediscovered while clearing out my old vinyl record collection. This album is mostly instrumental jazz-rock tracks with a very positive vibe. I love the mix of horns and percussion, as it reminds me of all kinds of bustling life in the city that I'd be missing while I'm all alone on the island.

What's it all about?

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

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

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

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

Next issue: Nat Pryce

Code Critique Competition 67

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

This issue's problem is also a bit of a design critique.

I've got a 'log' function (an externally provided function I can't change) that takes a **`char const *`** argument and I want to wrap it in a C++ layer so I can stream to it. The function itself is thread-safe and I want to be able to use streaming in multiple threads.

My approach here is to use a temporary helper object containing an **`ostringstream`** and build up the string in there. The helper object is created when the streaming starts, and is passed along the streaming operators until the end of the statement when it is destroyed. The destructor of the helper object passes the contents of the **`ostringstream`** to the log function. It seems to *nearly* work, but I'm getting some odd characters in the output. I found that adding an '**`&`**' (where it says: "**`/* Needed?: & */`**" seems to fix it, but don't know why. Are there any problems with this approach – or better ways to do the same thing?

(To give a bit of background, following a recent discussion in the ISO C++ standards meeting about destructors that throw exceptions, I was keeping a look out for examples of code where work was done in the destructor. That provided the initial input to this example, but it also raises questions about the lifetime of temporaries and the order of their destruction. Finally the code in the critique compiles with visual studio but not with g++ – making the change mentioned to add an ampersand fixes the compilation problem, so we have a compiler difference to consider too.)

The header file `log_wrapper.h` is shown in Listing 1. Listing 2 contains an example of its use: `test_log_wrapper.cpp`

Critiques

Chris Main <cmain@fastmail.fm>

Why are we going to the trouble of wrapping the log function in a C++ layer that provides a stream interface? Presumably to make it easy to use for programmers familiar with C++ streams. If so, then the interface ought to conform to the idioms of C++ streaming, otherwise it will be confusing or misleading.

The first thing I notice that surprises me is that the return type of **`logger::operator<<`** is not the **`logger&`** I expect, but helper. Returning a value when normally a reference is returned sets off alarm bells because of an experience I had a few years ago. I used a class that provided **`operator[]`**, but failed to spot that it was returning a value. I had assumed that it was returning a reference, and had a frustrating time tracking down the cause of some strange behaviour to the fact that I was operating on a temporary object rather than the object which I thought I had a reference to.

In this example it is not quite so bad because the value being returned is, essentially, a pointer, although I can only know this by examining the private part of the helper class, not the public interface.

The helper class, which should be an implementation detail, rather dominates the public interface. It looks like the author was aware of this and, judging by the **`friend`** declaration, tried at some point to hide it but failed.

The other non-idiomatic feature is that the logger appears to flush its output every time a semi-colon terminating a C++ source line is reached, and only then. Standard C++ streams do not flush at the end of every source

Listing 1

```
#include <memory>
#include <sstream>
// third party log function
extern void log(char const *);
// Add C++ style streaming to the log function
class logger
{
public:
  class helper
  {
    friend class logger;
  public:
    template <typename T>
    helper /* Need & here? */ operator<<(T t)
    { *oss << t; return *this; }
    helper() : oss(new std::ostringstream) {}
    // When helper destroyed log the message
    ~helper()
    {
      if (oss.get())
      {
        log(oss->str().c_str());
      }
    }
  private:
    std::auto_ptr<std::ostringstream> oss;
  };
  template <typename T>
  helper operator<<(T t)
  { helper h; return h << t; }
};
```

Listing 2

```
#include "log_wrapper.h"
// Dummy 'log' for testing
void log(char const *p)
{
  puts(p);
}
int main(int argc, char **argv)
{
  logger l;
  l << "Starting";
  l << "Testing number: " << argc;
  l << "Testing string: " << argv[0];
}
```

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

statement; they flush either when a **std::endl** is output to the stream or when **flush()** is called.

I ran the C++ hello world program under a debugger to find out what ultimately gets called by **std::endl**, and discovered that it was the virtual function **std::basic_streambuf::sync()**. So it appears that the STL designers have provided exactly the hook we need. All we need to do is derive a class from **std::basic_streambuf** and override the **sync()** function to call our log function and clear the buffer.

std::ostringstream uses **std::stringbuf**, which derives from **std::basic_streambuf** and provides a string buffer, so it is the obvious choice as the base class for our very own **logbuf**:

```
class logbuf : public std::stringbuf
{
public:
   virtual int sync()
  {
    std::string message = str();
    // Do any extra formatting of message here
    // eg handling embedded \0 characters and
    // adding or removing trailing new lines
    log(message.c_str());
    str("");  // clear the buffer
    return 0; // return default value;
              // (return -1 on error)
  }
};
```

Note that **std::stringbuf** provides a non-const **str()** member function as well as the more well known **const** one, and we can use that to clear the buffer.

At this point I was expecting to plug the **logbuf** into **std::ostring** stream, but discovered that you can't change the type of the stream buffer used by **std::ostring** stream. Lazily I turned to Google and found an article by Cay Horstmann originally published in the *C++ Report* in 1994 (http://www.horstmann.com/cpp/iostreams.html).

This showed that the rest of the plumbing I need is actually very simple:

log_wrapper.h

```
#include <ostream>
class logger : public std::ostream
{
public:
  logger();
  ~logger();
};
```

How's that for a clean public interface!The implementation is equally succinct:

log_wrapper.cpp

```
#include <log_wrapper.h>
#include <sstream>
namespace
{
  class logbuf ... // as above
}
logger::logger() : std::ostream (new logbuf)
{
}
logger::~logger()
{
  delete rdbuf();
}
```

Peter Sommerlad <peter.sommerlad@hsr.ch>

When compiling the code as given, gcc recognizes an error

```
../log_wrapper.h: In member function
  'logger::helper logger::operator<<(T)
  [with T = const char*]':
../main.cpp:8:   instantiated from here
../log_wrapper.h:36: error: no
  matching function for call to
  'logger::helper::helper(logger::helper)'
../log_wrapper.h:16: note: candidates are:
  logger::helper::helper(logger::helper&)
```

One can see from this 'strange' error message that the helper class obtains a non-regular copy constructor changing its origin. This is due to the **std::auto_ptr** member variable, since **auto_ptr** only has a **auto_ptr(auto_ptr&)** copy-ctor and not the regular using a const-reference as parameter.

Like with other pointer member variables (except for **shared_ptr**) one must define suitable own versions of **operator=** and the copy-constructor or deny the generation of these.

The latter is done, in classic C++ by declaring **operator=** and the copy-ctor as private members and never implementing them. In the new C++ standard due 2011, one would declare both as **=delete** instead, i.e.

```
helper& operator=(helper const&)=delete;
```

Nevertheless, the use of **auto_ptr** and dynamic memory allocation is completely unnecessary. A simple member variable of type **std::ostringstream** should suffice. However, we cannot copy stream objects simply.

Note: In C++0x one can use **std::unique_ptr** to get some of the effects of **auto_ptr** without the dangers. This relies on the so-called r-value references and "move"-constructors/assignment-operators. Explaining these mechanisms is beyond this critique.

With the proposed "fix" of returning a reference to the helper object the code compiles. Running lint (in our linticator plug-in – see Figure 1) shows:

- The messages 1732 and 1733 show us that there is a potential problem and suggest that an assignment operator and copy constructor might be required.
- The message 1793 shows also the suspicious use of a temporary.

The code tries to create a temporary helper object, that will be copied and the final one, should then release the collected characters to the **log()** function. I believe it is a design error to rely on such non-const-ref passed temporary objects. IMHO a better design would be to have the logger class' destructor to actually retrieve the string from the **ostringstream** and put it on the log. This way, the side effect is at least obvious from the code.

Thus a simple solution might be:

```
#ifndef LOG_WRAPPER_H_
#define LOG_WRAPPER_H_
#include <sstream>
```

Figure 1

Description	Resource	Path	Location
Warnings (3 items)			
1502: defined object 'l' of type 'logger' has no nonstatic data members	main.cpp	/scc66	line 7
1551: Function may throw exception '...' in destructor 'logger::helper::~helper(void)'	log_wrapper.h	/scc66	line 28
No return, in function returning non-void	main.cpp	/scc66	line 6
Infos (7 items)			
1732: new in constructor for class 'helper' which has no assignment operator	log_wrapper.h	/scc66	line 24
1733: new in constructor for class 'helper' which has no copy constructor	log_wrapper.h	/scc66	line 24
1793: While calling 'logger::helper::operator<<(char *)': Initializing the implicit object parameter 'logger::helper &' (a non-const reference) with a non-lvalue (a	main.cpp	/scc66	line 10
1793: While calling 'logger::helper::operator<<(int)': Initializing the implicit object parameter 'logger::helper &' (a non-const reference) with a non-lvalue (a tem	main.cpp	/scc66	line 9
1795: Template 'logger::helper::operator<<(<1>)' (line 19, file /Users/sop/workspaceCDTHelios/scc66/log_wrapper.h) was defined but not instantiated	scc66		line 0
1795: Template 'logger::operator<<(<1>)' (line 34, file /Users/sop/workspaceCDTHelios/scc66/log_wrapper.h) was defined but not instantiated	scc66		line 0
818: Pointer parameter 'argv' (line 6) could be declared as pointing to const	main.cpp	/scc66	line 11

```
// third party log function
extern void log(char const *);
// Add C++ style streaming to the log function
class logger {
  std::ostringstream oss;
public:
  ~logger()
  try {
    log(oss.str().c_str());
  }
  catch (...) { }
  template<typename T> logger& operator<<(T t)
  {
    oss << t;
    return *this;
  }
};
#endif /* LOG_WRAPPER_H_ */
```

- note that the destructor consists of a try-catch function body, so that no exceptions that might be thrown from calling log will be passed on. Without that guard a logger object destroyed while stack-unwinding from an exception happening will cause the program to terminate. Not a nice behaviour, if that is caused by an auxiliary function.
- returning by reference is usually unsafe, but returning **`*this`** from a member function is safe, even so lint still complains in main about this (as an information message)
- as a side effect of such a solution, the logger in the main function will only call **`log()`** once.

3. To get back to the previous behaviour of one **`log()`** call per line we need to change **`main()`** to avoid the explicit temporary but use a logger temporary per line instead:

```
int main(int argc, char **argv) {
  logger() << "Starting";
  logger() << "Testing number: " << argc;
  logger() << "Testing string: " << argv[0];
}
```

This will be no more expensive than before. The remaining thing might be that one can rename logger to log instead so that the code reads even more simply.

A more sophisticated implementation of the admired behaviour would be to implement our own **`streambuffer`** class, i.e. **`log_ostream_buffer`**, for log output that uses **`log(char const *)`** whenever the **`streambuffer`** should be flushed. This would allow us to create a log_stream object, as one might use any other **`ostream`**. However, doing so is overkill in this simple situation and explaining how to do it safely, beyond my time for this critique.

Huw Lewis < huw.lewis2409@googlemail.com>

Compilation

My compiler (GCC 4.4.5) complained about not being able to find a **`logger::helper::helper(logger::helper)`** constructor taking a helper parameter by value. This was down to the **`logger::operator<<`** returning the expression **`(h << t)`** that represents the **`helper::operator<<`** method, returning a helper object by value. I can't be precise about the nature of this error, but it is to do with the copy constructor for the helper class and its **`auto_ptr`** member that will reset itself to **`NULL`** on being copied (passing ownership of the pointee object away).

Oddly, adding **`'&'`** to the **`helper::operator<<`** method to make it return by reference fixes the compilation error. This removes the necessity for the copy operation coming out of **`helper::operator<<`**, but does not remove the copy operation from the parent **`logger::operator<<`** method so I remain a little puzzled by this. Could it be to do with inline template methods?

A more satisfactory (or explainable) solution to the compilation issue is to make the **`auto_ptr`** data member **`'mutable'`** so that it can be nullified while the object is **`const`**, and to explicitly add a copy constructor.

Object lifetimes

To understand what is going on here, let us discuss the object lifetimes. The logger object **`'l'`** is constructed and survives until the end of the function (and program in this case).

The **`logger::operator<<`** method returns a helper object by value. In typical usage (as per the test harness) this helper object is a 'temporary' anonymous object that is destroyed at the end of the line. The destruction of this object (or objects!!) is what initiates the sending of the log data to the external log function. This is a neat design choice as messages are sent to the log at the correct time. If, for example, it were the destruction of logger that sent the data to the log then many messages built up over a potentially long (maybe including blocking operations) lifetime.

In the original version, the **`helper::operator<<`** also returns a helper object by value which has the effect of creating additional copies of the helper on each streaming operation. The helper class uses an **`auto_ptr`** to manage the lifetime of the **`ostringstream`** internal object so ownership is passed along from copy to copy and only the final temporary object does the destruction processing, passing the data to the external log function.

When the **`&`** is added, the return type is by reference so these additional copies don't occur. We have the one logger object and a single helper (anonymous temporary) object.

Odd characters in output?

The question says that the original version (which I couldn't compile) prints some odd characters in the output. I can't see the reason for this behaviour (and can't reproduce it). How could erroneous characters enter the stream? I can see situations that could lead to crashes where a helper object attempts to be use the **`auto_ptr`** to the stream after it has passed on ownership to the next helper copy.

We can disregard threading issues as the (external) log function is thread safe. Changing the return type to reference wouldn't affect that anyway.

Could there be some strange optimisation issues with all of the temporary helper copies?

Design comments

I think that the designer/coder has got fairly close to a good solution with this already, but we can always improve. I like the way that the logger attempts to have stream semantics and I especially like the way that the data is sent to the external log on destruction of the temporary helper objects at the end of the line.

However, I have read conflicting information on the lifetime of temporary objects. One source claimed that the C++ standard does not define the lifetime of temporary objects, therefore it is compiler dependent and code that depends on the lifetime of such objects might not be portable. Another source states that temporary objects exist until the end of the full statement in which they are created. [Ed: there was ambiguity in compilers in the 1990s but the C++ standard *does* give clear guidance over the lifetime of temporaries.]

There is another burning issue around the idea of using the temporary helper object. Streams are types that simply aren't meant to be copied. The **`std`** stream types explicitly don't provide copy constructors for that very reason. Our technique of returning a helper object by value from **`logger::operator<<`** skirts around this by using the **`auto_ptr`** to pass ownership of the same underlying stream to the next helper copy rather than creating a new stream copy. This leaves open the possibility for some poor or malicious coder to create a helper copy and use the original after a copy has been made – the **`auto_ptr`** to the stream is now invalid!! You could code defensively to prevent a crash, but these niggles are telling me that it isn't the best design choice.

The corrected **`helper::operator<<`** returns a reference to the helper object, allowing the chaining of streaming operations. As helper is not

derived from a **std** stream class, it will not play nicely with stream manipulators (e.g. **hex**, **endl**, **setfill**, **setw** etc) and so we aren't able to perform any fancy formatting. We could make **logger** and helper derive from **ios_base**, but that is likely to raise many more complications.

Extra requirements?

Logging seems like such a simple concept, but in reality this is rarely the case. Consider the following:

- we'd like to remove the logging code from release builds
- we'd like to run the system at varying levels of logging (info, warnings, errors, etc).
- we'd like some extra metadata logged with each message e.g. timestamps, thread id etc.

How would our previously reasonably elegant design handle these issues?

Alternative designs

One fundamental requirement I have experienced many times (especially in high performance embedded systems) is to remove all logging code from release build configurations. The most effective method to do this is something we usually try to avoid – macros. They are good for some things. For example:

```
#define LOGGING_ENABLED
// or define the symbol via the Makefile
#ifdef LOGGING_ENABLED
// the real implementation
#define WRITE_LOG(x) blah....
#else
#define WRITE_LOG(x) // defined as nothing
#endif // LOGGING_ENABLED
```

Now all logging code will be pre-processed away when **LOGGING_ENABLED** is not defined.

Another nice feature (inspired by the original implementation) is to use streaming semantics inside the macro argument e.g. **WRITE_LOG_STREAM("some string" << stringVariable << std::hex << someIntVariable)**, where:

```
// log wrapper function
void log_wrapper(const std::string& out);
...
#define WRITE_LOG_STREAM(x) \
{ std::ostringstream tempStream; \
tempStream << x; log_wrapper(x.str());}
#define WRITE_LOG(x) log_wrapper(x);
```

Where different types or levels of logging are required, it is fairly simple to define variants with additional arguments e.g.:

```
void log_wrapper(const std::string& out,
  LogLevel level);
...
#define WRITE_LOG_LEVEL(level, x) \
  log_wrapper(x, level);
```

By keeping a simple interface (facade) made up of a small number of helper functions and macros, the application code is kept simple and the flexibility is retained to vary the implementation and behaviour as needed.

Commentary

This critique was inspired by some logging frameworks and, although I agree with Chris Main that we expect **operator<<** to only flush when **std::endl** is used (or **flush()** is called), there is a reason to avoid this for a logging stream.

The problem comes when a program aborts; if this occurs then any un-streamed data in the log buffers may contain the vital piece of information that will help analyse the failure. If the logger auto-flushes at the end of each line then all useful data will have been flushed out.

However C++ does not provide a way to detect the end of the statement other than by inference when temporary objects are destroyed. Hence the code provided uses the destructor of the last helper object to write the complete message to the log. The problem with using this approach for implementation is the 'double destructor' problem: should the destructor of the temporary object throw an exception while unwinding from *another* exception then the program will abort by calling **std::terminate()**. This is a bad end for a program!

The general rule is not to throw from destructors – but it is easy to break this rule, as we do here. Consider what might happen when logging a couple of largish messages in a low memory situation: when we try to append the second message to the **ostringstream** there isn't enough memory to increase its internal buffer, so a **bad_alloc** exception is thrown. This causes stack unwind, so temporary objects are destroyed – including the logger helper object. The destructor of the object calls **str()** on the **ostringstream**, which also fails because of the low memory and so the program aborts.

The winner of CC 66

There was some good discussion in Huw's critique and although I have much sympathy with a macro solution I am always a little concerned at the possibility of unexpected behaviour. I'm not quite sure what problems there were with **hex**, **endl** and other manipulators: they appear to work fine.

Peter made good use of lint to try and explore the problem although I was a bit concerned at the large number of messages generated that were simply wrong: there should be no need to write an assignment operator or copy constructor in this case as the **auto_ptr** was supposed to handle the ownership in the compiler-generated defaults. However his solution, using a temporary logger object and thus removing the need for a helper class completely, seemed to be a step in the right direction. The addition of the catch statement to the destructor also fixes the potential for program abort on a 'double fault'.

Chris pointed out the helper class dominated the public interface of the logger class: this is true; a forward reference of

```
class helper;
```

would allow moving the actual implementation to later in the logger class and make it clearer. I liked his attempt to use a **logbuf** class although his

Listing 3

```
#include <string.h>
#include <string>
#include <iostream>
std::string castToString(wchar_t * wideStr)
{
  std::string str;
// This is what I want, but it won't compile:
//  str = (std::string)wideStr;
// This compiles, but I just get "H":
  str = (char*)wideStr;
// This compiles too, with the same output:
  std::wstring wstr;
  wstr = wideStr;
  str = (std::string)(char*)wstr.c_str();
// This is nearly there I think:
// but I now just get "H e l l o"
  str = std::string((char*)wstr.c_str(),
        wstr.size());
  delete wideStr;
  return str;
}
int main()
{
  wchar_t * source = new wchar_t[12];
  memcpy(source, L"Hello world", 24);
  std::string str = castToString(source);
  std::cout << str << std::endl;
}
```

Bookcase

The latest roundup of book reviews.

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

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

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

Jez Higgins (jez@jezuk.co.uk)

Software Requirements & Specifications

By Michael Jackson, published by Addison-Wesley, ISBN 0-201-87712-0

Reviewed by Paul Floyd

Highly recommended

Not the Michael Jackson that made the top-selling pop album, nor the one that writes about drinking beer. This is the software method Michael Jackson. The subtitle of the book is 'a lexicon of practice, principles and prejudices'. I would have added 'philosophy' in there somewhere, as this is the software book with the most pronounced philosophy bent that I've ever read. Just as well that's a good thing in my eyes. Polya, Logical Postivism, Bertrand Russel and Karl Popper all get mentions. Jackson makes a fair bit of use of predicate logic, which might be off-putting if you are significantly mathematically challenged.

Looking at the bibliography, much of the material is from the 60s to the 80s. Not necessarily a bad thing, but it does somewhat place Jackson with the ideas of structured programming (which, as top-down design, he frequently denigrates).

One thing I particularly enjoyed was that there is a fair bit of humour and dry wit. Not too much, but enough to give me a few chuckles, in particular the item on 'Brilliance', p. 20.

I thought that there was plenty of sound advice, like concentrate on the problem, not the solution, and do your thinking at the right level of detail ('span', as Jackson calls it). There is a bit of repetition regarding Jackson's 'Frames' method, but he doesn't bang on about it incessantly.

I haven't read that many books about requirements and specifications, but they have tended to concentrate on things like use cases and 'requirements gathering'. This book isn't like them, and in fact, not like any other book I've read. As a bonus, it's only just over 200 pages long, so was quite easy to digest.

Reflections on Management

By Watts Humphrey, published by Addison Wesley, ISBN 978-0-321-71153-3

Reviewed by Paul Floyd

Recommended

To give the book its full subtitle 'How to Manage Your Software Projects, Your Teams, Your Boss, and Yourself'. Quite an ambitious project.

For those that don't know about Watts Humphrey, he's the man behind much of CMMI. He earned his stripes at IBM, managing the development of OS/360.

The book has something of a *Readers' Digest* of the other works of Humphrey (on PSP – Personal Software Process and TSP – Team

Bookshops

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

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

Code Critique Competition (continued)

simple stream (using Cay Horstmann's article) is slightly *too* simple as it loses the thread safety of the original code.

I found it hard to choose the winner, but eventually decided to give this issue's prize to Peter.

Code Critique 67

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

Can anybody help me to cast wide characters to an stl string? I can handle single characters successfully like this:

```
std::string str;
wchar_t wch = L'X';
str += (char)wch;
```

but I can't seem to get the syntax right for doing a whole array of them. Here's a program showing what I've tried (Listing 3).

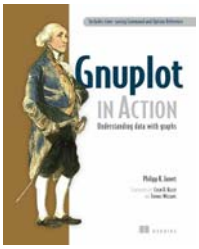
You can also get the current problem from the accu-general mail list (next entry is posted around the last issue's deadline) or from the ACCU website (http://www.accu.org/journals/). This particularly helps overseas members who typically get the magazine much later than members in the UK and Europe.

Software Process). That's not a bad thing, as it makes the book refreshingly short and accessible. The tone throughout is brusque, positive and upbeat. I don't think that Humphrey has much time for people that have doubts and hesitate. Issues are tackled head on, usually with a step-by-step plan.

Broadly, the themes covered in the book are software quality, making yourself and your team efficient, and managing yourself and your boss. Personally, I'm not entirely convinced by the software-process-as-industrial-process thesis, but clearly it does have merits. The book contains plenty of plain common sense, like 'define your objectives' and 'always start with a plan'. What I found most interesting (and true to life) in this book were the human insights. For instance, what managers at various levels expect and aim for: a Vice President with strategic long-term goals compared to a line manager with tactical short-term goals.

Final word, I found the book interesting enough for me to get a copy of *PSP(sm): A Self-Improvement Process for Software Engineers* by the same author.

Gnuplot in Action



By Phillip K. Janert, published by Manning, ISBN 1933988398

Reviewed by Giuseppe Vacanti

Recommended

Gnuplot is a plotting program, with some data analysis and manipulation functionality. It is completely command-line based, and because of this it can be easily scripted. This feature makes Gnuplot very appealing if a large amount of data must be routinely processed in order to produce standard plots.

This is a very good book that covers all the aspects on Gnuplot, from the elementary to the complex ones. As the subtitle - Understanding data with graphs - suggests, the book is more than just a guide to the program: the author also ventures into explaining why one would want to plot data, and what they might learn from it.

For those less at ease with all the terminology related to the graphical representation of data, the author takes the time to introduce the more important concepts. For instance, in chapter three the facilities related to logarithmic plots are introduced. Logarithmic plots are not however something that one comes across very often outside of scientific and technical fields, so the author explains the mathematics behind this type of graph. Sections such as this one are clearlyidentified and can be easily skipped if one is already familiar with their content.

The subject matter is grouped in four parts. In the first part we learn how to make increasingly more complex plots, and we are introduced to Gnuplot's data manipulation capabilities. At this stage we are not concerned with the details of how our graph looks like: the default look is sufficient to gain insights into the data.

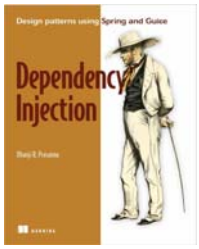
Having discussed how to make a plot, in the second part the author dives into the details of how to customize the look of our plot. Like any good plotting program, Gnuplot allows one to control almost every pixel on the canvas, and we learn about symbol and line styles, axes, legend boxes, multiple axes, and more. This part is appropriately titled Polishing: the basic plot we have, and now we want to make it look good in every detail.

Whereas chapters in the first two parts must be read one after the other, chapters in part three (Advanced Gnuplot) can be read according to interest or need. We learn about three-dimensional plots, color management, curve fitting, non-cartesian plots, fonts and output formats, and scripting (including how to plug Gnuplot behind a CGI script!). By the end of part three we know all there is to know about Gnuplot.

As hinted to earlier, an underlying theme across the book is how graphs are a fundamental tool in the understanding of data, and many of the examples make this clear. In the fourth and final part of the book the theme of graphical analysis is central, and now we explore how to approach data analysis and interpretation, and how Gnuplot can help us achieve an extra level of understanding. Here the author does not shy away from some math and more complex topics, but these can be skipped, as they do not introduce any new functionality.

In summary a very good book describing a very good piece of software.

Dependency Injection: Design patterns using Spring and Guice



By Dhanji R. Prasanna, published by Manning, ISBN 978-1-933988-55-9

Reviewed by Paul Grenyer

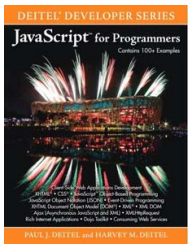
This is a great book. When I first saw it on the ACCU review list I requested it out of pure curiosity. Having used dependency injection for some time I was intrigued to find out how what appeared to be such a straightforward subject could be stretched into a 300+ page book. I was also pleased to find out that the book came with a free PDF download, so it became the first book I read on my Kindle.

Dependency Injection is quite a simple idea, but there are pitfalls and plenty of things, such as naming and scope, that must be considered when using it. Dhanji goes into all of these in a lot of detail and I suspect even seasoned dependency injection users will learn something. He also gives examples in both Spring and Guice all the way through and finishes with a complete web application in Guice.

I was also pleased to read about Dhanji's deep dislike for the Singleton pattern, but he did spend several pages explaining the problems. Although this was good to read, it isn't really related directly to dependency injection and could probably have just beenreferenced. There are quite a few topics in the book when Dhanji goes off on a tangent. It's all good stuff, but not particular relevant attimes. The only place where he gets it a bit wrong is his description of the behaviour of finalize.

If you are interested in dependency injection, read this book!

JavaScript for Programmers



By P.J. Deitel and Harvey Deitel, published by Prentice Hall, ISBN: 978-0137001316

Reviewed by Ian Bruntlett

This is an excellent book to get to grips with internet programming. Like all of computing, internet programming is constantly evolving, constantly making books out of date. There are plenty of source code listings in the book, with certain lines highlighted for those things that are relevant to the chapter's topic.

Most of this book's chapters have a 'Web resources' section at the end. The book is useful for programmers who know about the web (e.g. HTML) and want to get to grips with JavaScript. It starts with an introductory chapter which ensures that the reader is up to speed with the basics. Chapter 2 is an introduction to XHTML and provides useful links like validator.w3.org.

Chapter 3 details Cascading Style Sheets (CSS) that are used to separate structure from presentation which makes it easier to change the appearance of a website by just selecting a different style sheet. I was particularly interested in its coverage of 'Media Types' where web pages adapt to the type of the device used to display content such as a computer screen, a printer, a MID or, for challenged users, braille or aural media types.

Chapters 4–9 cover JavaScript (Introduction (4), Control Statements (5,6), Functions (7), Arrays (8), Objects (9), Document Object Model aka DOM (10), Events(11).

Chapter 6 is fairly routine except that statement blocks can have labels which in turn are referred to in break and continue statements.

Chapter 7 provides a good JavaScript example - implementing the Craps dice game.

Chapter 8 describes arrays in JavaScript. The only new thing I noticed was that when a value is assigned beyond the current size of an array, that array is resized. And functions can be stored in arrays as well, I am told. Chapter 9 discusses JavaScript objects – like a standard library for

View From The Chair

Hubert Matthews
chair@accu.org

At the next conference our current committee secretary, Alan Bellingham, will be standing down after ten years. Outside the hallowed sanctum of the committee, the general membership may not realise what the secretary does or appreciate how much he can help or hinder the smooth working of the committee (and therefore the ACCU itself). Alan has provided a much-needed sense of calm continuity as a number of chairmen have arrived and departed. I'm sure, like me, my predecessors would all wish to thank him for his able assistance. Alan kindly deferred his departure for a year to avoid a simultaneous change of secretary and chairman, for which I'm most grateful. Of course, this leaves a golden opportunity for someone to take over the secretary's role and to make it their own. The duties are not onerous and the key capability required is to be organised. Anyone who wishes to know more should contact Alan or myself directly. If no-one is forthcoming I shall have to mount a charm offensive; you have been warned.

On the accu-general mailing list there has recently been a continuation of the discussion of the role of the ACCU and how it relates to other organisations in computing such as the BCS and the IET. I am happy to see this discussion continue and the level of interest and commitment to the ACCU is very encouraging. We certainly do need to continue to think where we want the ACCU to go in the next few years. Our industry is changing in many ways and we must be a part of that change, and hopefully be at the vanguard of it rather than being pulled along reluctantly or told what to do by others. Our industry is a young one and it is still forming, still working out how the myriad pieces fit together. Although we are in the first flushes of hopeful and expectant youth compared to medicine or other professions we have achieved a lot in a short space of time. That time, however, has been long enough to last a lifetime for some. For instance, within the last few months we have lost two notable names in our industry – Sir Maurice Wilkes and Watts Humphrey – so it is probably a good time to reflect on what they achieved and what we can learn from them and their lives. These two pioneers had vision and the drive to see it through. From the earliest computers through to the dawn of networking with the Cambridge Ring, or as one of the founders of software engineering, their discipline and commitment remind us of what we can all achieve with focus and rigour. We won't all reach the levels of these men but we can all improve what we do and how we do it in our own environments. I believe that the things that inspired and motivated these men are the same values that drive ACCU members and their desire to create new solutions, learn new techniques and hone their skills. We should not be daunted by them but rather inspired by them –JAN 2011 they set a pathway that we can and should follow, both as individuals and as a group. We cannot repeat what they did; we must find a new way. As software designers, creating novel solutions and envisioning architectures for systems is what we are good at and what we enjoy. All we need do is to apply these skills to our industry and to the ACCU itself instead of to the software we write. Now there's a challenge worth taking on and one that the membership is ideally placed to do!

Bookcase (continued)

JavaScript – Math / String / Boolean / Number / Document / Window and is a very interesting chapter.

Chapter 10 covers the Document Object Model (DOM) and shows the DOM tools that can be added to IE or FireFox. Chapter 11 JavaScript: Events explains how web pages can react to events in the browser. Chapter 12 covers XML and RSS. XML is the eXtensible Markup Language.

Chapter 13: Ajax-enabled Rich Internet Applications (RIA) are introduced, allowing a developer to develop lots of Cool Things. The obligatory appendices are here, covering a) XHTML special characters, b) XHTML colours and c) JavaScript operator precedence chart.