The magazine of the ACCU

www.accu.org

{cvu}

Volume 23 • Issue 2 • May 2011 • £3

{cvu}
Volume 23 Issue3
May 2011
ISSN 1354-3164
www.accu.org

Features Editor
Steve Love
cvu@accu.org

Regulars Editor
Jez Higgins
jez@jezuk.co.uk

Contributors
Pete Goodliffe, Paul Grenyer, Richard Harris, Yechiel Kimchi, Chris O'Dell, Roger Orr, Seb Rose

ACCU Chair
Hubert Matthews
chair@accu.org

ACCU Secretary
Alan Bellingham
secretary@accu.org

ACCU Membership
Mick Brooks
accumembership@accu.org

ACCU Treasurer
Stewart Brodie
treasurer@accu.org

Advertising
Seb Rose
ads@accu.org

Cover Art
Pete Goodliffe

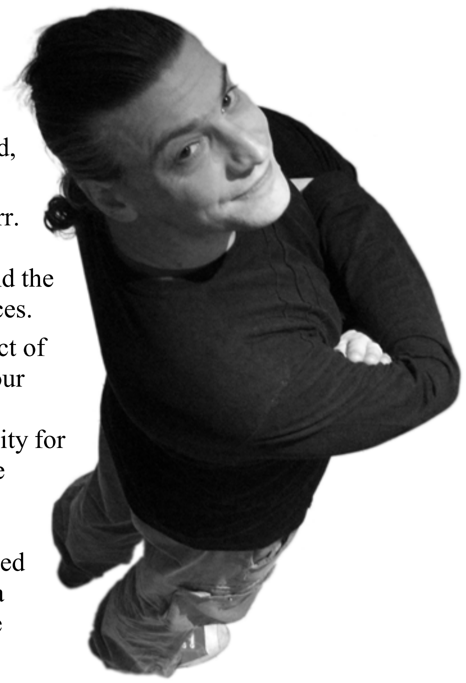
Repro/Print
Parchment (Oxford) Ltd

Distribution
Able Types (Oxford) Ltd

Design
Pete Goodliffe

accu

Coincidence and convergence



As I write this, I'm in the last stages of preparing a talk for the ACCU 2011 conference. By the time you read this, of course, the conference will be finished, and maybe you attended and if you did, I hope you enjoyed it!

Anyway, this year I'm sharing a session with Roger Orr. This came about because we each proposed our own talks on such a similar topic that, due to timetabling and the sheer number of proposals, we felt we should join forces.

As it happens, we each proposed a talk about the subect of Equality, and funnily enough, at least part of each of our inspiration came from a talk given some time ago by Angelica Langer (possibly best known to this community for being co-author of *C++ IOStreams and Locales*). She spoke on the subject of Equality in Java (I don't know whether Roger and I saw the exact same talk, but the topic was the same). This was a sufficiently complicated topic, especially for C++ programmers coming to Java for the first time, that it fully justified the length of the session.

Both Roger and I thought that it was still sufficiently difficult – not least because the problems are different for each langauge – to deserve another session.

That a talk given by someone on a topic that certainly wasn't *directly* relevant to me (at the time) would one day affect my thinking in general about the problem isn't that surprising. That it so affected at least two people the same way to the extent that they *independently* propose their own talks on the subject, several years later, is perhaps unexpected. I suppose my point is this: funny how things work out, isn't it?

Steve

STEVE LOVE
FEATURES EDITOR

The official magazine of ACCU

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

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

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

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

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

DIALOGUE

REGULARS

FEATURES

SUBMISSION DATES

C Vu 23.3:	1st June 2011
C Vu 23.4:	1st August 2011
Overload 104:	1st July 2011
Overload 105:	1st September 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

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

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

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

COPYRIGHTS AND TRADE MARKS

Some articles and other contributions use terms that are either registered trade marks or claimed as such. The use of such terms is not intended to support nor disparage any trade mark claim. On request we will withdraw all references to a specific trade mark and its owner.

By default, the copyright of all material published by ACCU is the exclusive property of the author. By submitting material to ACCU for publication, an author is, by default, assumed to have granted ACCU the right to publish and republish that material in any medium as they see fit. An author of an article or column (not a letter or a review of software or a book) may explicitly offer single (first serial) publication rights and thereby retain all other rights.

Except for licences granted to 1) Corporate Members to copy solely for internal distribution 2) members to copy source code for use on their own computers, no material can be copied from C Vu without written permission from the copyright holder.

A Game of Divisions

Baron Muncharris sets a challenge.

Greetings Sir R-----! I trust that I find you in good spirit?

Will you join me in a draught of this rather fine Cognac and perchance some sporting diversion?

Good man!

I propose a game that ever puts me in mind of an adventure of mine in the town of Bağçasaray, where I was posted after General Lacy had driven Khan Fetih Giray out from therein. I had received word that the Khan was anxious to retake the town and been given orders to hold it at all costs.

I had at my disposal three divisions of the Empress' own Hussars with which to fend off the Khan's impending assault. Unfortunately I knew not whether he intended to concentrate his forces at the north or at the south gate.

This placed me in something of a quandary; entire, I was certain that my company should prove more than a match for his, but divided they might easily be overwhelmed.

After much deliberation I decided to place my trust in fate and directed all my men to defend the north gate, whilst I alone stood guard over the south.

To my very great fortune, the Khan brought the greater part of his company to bear upon the south gate, affording me an excellent morning constitutional!

Modesty forbids me from observing that the lesson I gave them that day in no small part contributed to their ultimate defeat at Qarasuvbazar.

But I fear I digress! We should most assuredly be better served if I described the manner of play!

Here before you I have placed an ace of spades, which shall stand for the north gate, and an ace of hearts, which shall stand for the south.

At each turn you must wager three coins from your purse upon these gates. You may divide the coins between them howsoever you wish; you may place your coins in whole or in part, it is of no consequence.

I shall then toss this here coin; if it comes up heads then the north gate shall win the turn, else it shall be the south.

Those coins you wagered upon the winning gate shall be returned to you and I shall add to them a bounty of equal size. Be warned, however, that I shall have those wagered on the losing gate for my own purse!

Honour compels me to add that the coin is not fair; on the average it comes up heads nine times out of twenty and tails eleven times out of twenty. Or was it the other way around? Now I think on it, I have quite forgot!

No matter! The game shall last six turns and cost you one fifth part of a coin to play.

When I described the rules of play to that oafish student acquaintance of mine he, true to form, commenced babbling on about some utterly inconsequential topic. As I recall it was some hospital tragedy involving triangular pastilles of all things! I should say that the only hospital he should concern himself with is the Bethlam Royal and of his not being admitted therein forthwith!

But let us not tarry upon his ridiculous concerns. Come, take another draught and think upon how you may best hedge your bets! ■

Listing 1 is a C++ implementation of the game.

Listing 1

```
bool
bias()
{
  return
     double(rand())/(double(RAND_MAX)+1.0) < 0.5;
}
bool
toss()
{
  return
     double(rand())/(double(RAND_MAX)+1.0)
     < 0.55;
}
void
play()
{
  const bool bias_north = bias();
  double balance = -0.25;
  for(unsigned i=0;i!=16;++i)
  {
    std::cout << "Round " << i+1 << ": ";
    std::cout << "Balance
       = " << balance << std::endl;
    double north_coins = -1.0;
    while(north_coins<0.0 || north_coins>3.0)
    {
      std::cout <<
         "How many of your 3 coins will ";
      std::cout <<
         "you wager on the north gate? : ";
      std::cin  >> north_coins;
      if(north_coins<0.0 || north_coins>3.0)
      {
        std::cout << "I beg your pardon?" <<
           std::endl;
      }
    }
    balance -= 3.0;
    const bool north
       = bias_north ? toss() : !toss();
    if(north) balance += 2.0*north_coins;
    else      balance += 2.0*(3.0-north_coins);
    if(north) std::cout << "The north ";
    else      std::cout << "The south ";
    std::cout << "gate wins!" << std::endl;
  }
  if(balance>=0.0)
     std::cout << "You won "  <<  balance;
  else if(balance<0.0)
     std::cout << "You lost " << -balance;
  std::cout << " coins!" << std::endl;
}
```

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.

Testing Times

Seb Rose shows that unit tests are for everyone.

Like most people who have been exposed to popular western cinema I know that the right place to start is 'at the very beginning'[1]. You've got to understand, though, that the beginning isn't necessarily where you think it should be – it's where I want it to be. And so, in time honoured Hollywood tradition, I'm going to start this tale at the end. With a question.

Imagine you have been working on a software project. How would you determine that the current phase of the project had completed successfully?

Assume that each phase delivers something useful to your customers. We should also assume a broad definition of customer that includes not only a fee paying public, but a wage paying organisation, down stream colleagues and implementers of future maintenance and enhancement requests.

That's a lot of people to satisfy. Did we just start thinking about them at the end of the project?

'Not at all,' you say, 'we thought about these customers all the way through the project. No one in their right mind would leave all this hard thinking till the end of a project. When the deadline is approaching and the feature set is incomplete and that intermittent defect is sucking up all our time, the last thing we want to think about is whether the product is going to be easy to support.'

You're right. Of course, you're right, but lots of organisations work this way. They may have product management feature lists, a documentation department, an installation development team, quality indicators and serviceability specifications. They may even work in parallel with product development. There may even be (gulp) regular, meaningful communication between all these teams, but in the end those last weeks and months are typically frantic and fraught with compromise and mitigations.

Now, back to the question – how do you measure success? The simple answer is that all your customers should be satisfied, but how can you tell that they will be satisfied BEFORE you release the product?

One part of the answer is early feedback, beta programmes and the like, but in this article I'm going to talk about testing. Acceptance testing, integration testing, unit testing. Specifically, automated testing, using the tests to drive the product development – outside-in.

Acceptance Test Driven Development

Here we are at the beginning of the project. What should the development team have to get them going? A detailed, signed off requirements document? A prioritised backlog of user stories? A vision statement from the corporation's Evangelist Tzar? We all have our own preferences, but the reality is you take what you're given and start from there.

At this point I'd like to introduce a remarkable book that came out in 2009 – *Growing Object Oriented Software Guided By Tests* [2]. It's remarkable in many ways, not least because of its utilitarian 'does what it says on the tin' title, so if you haven't read it yet I urge you to borrow it from your local public library (if you still have one). The book covers a lot of ground, and the design guidelines are directed at OO development. However, a lot of the book describes a test-driven development (TDD) feedback loop – see Figure 1 – that is not only applicable to OO development. [3]

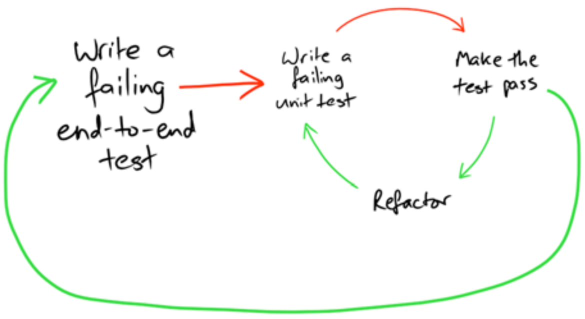


Figure 1

The authors make use of Cockburn's idea of starting a development project with a 'walking skeleton' [4], which is (my highlights):

> an implementation of the thinnest possible slice of real functionality that we can **automatically build, deploy and test end-to-end**

This addresses some of the important barriers to automated testing up front, letting us use automated tests (at all levels) to drive all subsequent development. We can now write acceptance tests that can validate when a piece of functionality is delivered, and this is known as Acceptance Test Driven Development (ATDD).

I'll go into more detail about the feedback loop used in all levels of TDD a little bit later, but the point here is that, using ATDD, product owners can collaborate with the development team to produce acceptance tests BEFORE the code implementing the features is developed. These acceptance tests serve as design and documentation of features and (as long as they are run regularly) CANNOT get out of sync with the code.

Over the years, a lot of work has gone into developing frameworks that allow acceptance tests to be expressed in a way that is consumable by business people as well as development. One of the earliest examples I'm aware of is FIT [5], which has been made more interactive by Fitnesse [6], but there are many others (notably Cucumber [7]). The underlying thrust is to allow the creation of a Domain Specific Language (DSL) that can be used to write tests that describe (in a readable way) how to test that a feature works as intended. The framework then translates the DSL into actions that execute the tests.

Don't, however, get distracted by the ATDD tools – which can lead to an ATDD cargo cult. The value of ATDD is in improving communication between business people and development. The whole subject is given a much more detailed treatment in Gojko Adzic's book *Bridging the Communication Gap* [8]. And, as Elisabeth Hendrickson said recently on her blog [9]:

> Starting an adoption of ATDD with the tools is like building an arch from the top. It doesn't work.The tools that support ATDD – Fitnesse, Cucumber, Robot Framework, and the like – tie everything together. But before the organization is ready for the tools, they need the foundation. They need to be practicing collaborative requirements elicitation and test definition. And they need at a bare minimum to be doing automated unit testing and have a continuous automated build system that executes those tests.

Test Driven Development

A few weeks ago I went to the 'Simple Design and Test' Conference in London. It wasn't particularly well attended (possibly because it was on a weekend or maybe it wasn't well promoted) which is a shame, because

SEB ROSE

Seb is a software contractor in some of the most hostile environments, including banks, pensions and manufactuers. He works with whatever technology is prescribed by the client – mainly C++, .NET and Java.

unnecessary complexity, whether intentional or accidental, makes life difficult.

It's not easy to identify when complexity is unnecessary, nor is it easy to simplify something once it has been created. This is not news. The XP community has long followed the precept of YAGNI (You Ain't Gonna Need It) which follows Ron Jeffries' advice [10]:

> Always implement things when you actually need them, never when you just foresee that you need them.

Most of us will also be familiar with the pragmatic advice of KISS (Keep It Simple Stupid) which advises us that simplicity should be a key goal in design, and that unnecessary complexity should be avoided.

Why is it then, that software often seems to get complicated and messy? Are there techniques or practices that can help us keep our house in order?

My answer is: 'Yes. TDD.'

Those of you not familiar with TDD might find it odd that I'm motivating a practice with 'Test' in the title by saying it will help our design. So, at this point let me alter the meaning of the acronym and claim that TDD stands for 'Test Driven Design'. (I'm not alone in thinking that this is a reasonable thing to do!)

Whichever expansion of TDD you prefer, the TDD process can be simply stated as:

- Write a failing test
- Write just enough code to make it pass
- Refactor
- Repeat

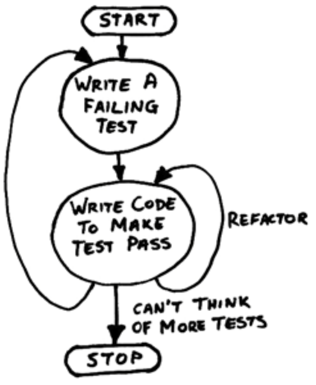
Represented graphically, it looks like Figure 2.

This is deceptively simple – like so many techniques in life, it takes skill that needs to be developed through repeated application of the principles. For starters, you should only write enough code to make the test pass. Resist the temptation to write extra code; don't think ahead to the next test. This seems counter-intuitive, if not down-right wasteful.

Bob Martin recently blogged about his 'Transformation Priority Premise'. He uses the example of the Bowling Game Kata [11], where the first test checks that if you don't hit any skittles, the score is 0. At this point his implementation of the `score()` method simply returns 0. Surely this is brainless:

> At this point the programmers in the class roll their eyes and groan. They clearly think this is dumb and are frustrated that I would be telling them to write code that is clearly *wrong*.
>
> What I have begun to discover is that returning zero *is not nearly so brainless as it looks*. Not when you put it in the appropriate context.
>
> When we use TDD, our production code goes through a sequence of transformations. I used to think it was a transformation from stupid to intelligent. But I've begun to see that this is not the case at all. Rather, the code goes through a sequence of transformation from *specific to generic*.

Figure 2



> Returning zero from the `score` function is a specific case. But the case is in the correct form. It is an integer, and it has the right value. Therefore the *shape* of the algorithm is correct, it just hasn't been generalized yet.

It's also important not to skip the refactoring step. Remove duplication as it arises. Extract, decompose, simplify. This applies to unit testing in general not just TDD, but it's crucial that you keep your code and your tests clean as you go, because **there is no later**.

Developers often feel very uncomfortable when they start developing the code using tests. We're used to architecture and design and specification. TDD certainly de-emphasises the importance of up-front design and 'paper' documentation and instead focuses on **emergent** design. That doesn't mean you don't need to consider architectural issues. On the contrary, as you write the next test, implement the code and refactor you are making **intentional** design decisions all the time.

There's a great session on QCon where Jim Coplien and Bob Martin discuss whether architecture is necessary up-front, and accept that you might need around a half an hour architecture discussion before embarking on a medium sized telecoms product. [12]

And just in case you think that you can't practise TDD because it isn't supported in your environment, have a look at the 'Bowling Game Kata in C' [13].

Unit tests

Good Unit Tests are the foundation of these practices. They've been around for decades, and lots has been said by the great and the good [14]. But if it's worth saying once, it's worth saying again…

First let's get one thing straight, from Michael Feathers [15]. A test is NOT a unit test if:

- It talks to the database
- It communicates across the network
- It touches the file system
- It can't run at the same time as other unit tests
- You have to do special things to your environment (such as editing config files) to run it

The reason for these restrictions is that unit tests need to run quickly and reliably. Developers practicing TDD will run them every few minutes and automated builds will need to run them unattended. Tests that have dependencies on the environment have ways to fail that have nothing to do with the behaviour under test and I wouldn't include them in a unit test suite. Your mileage may vary.

In the tradition of coining '-ities', I propose the following 5 for unit testing (along with a cast iron rule)

- Testability
- Necessity
- Granularity
- Understandability
- Maintainability

Unfortunately, I can't make a nice acronym out of these: MUNGT, TeNGUM. All suggestions gratefully received.

Testability

Writing unit tests is hard. People argue about what constitutes a unit. Again, I have my opinions, but as long as they pass the Feathers test (above) I'm not going to get into that argument here. But… you do need some idea of unit, and when writing a test, you need to test just the unit-under-test. If the unit-under-test depends on other units you can get into a huge muddle trying to get those dependencies into a suitable state. This is when you need to avail yourself of whatever language feature is available to implement a **seam**.

The seam is a concept introduced by Michael Feathers [16] that he describes as:

> a place where you can alter behavio[u]r in your program without editing in that place

How you alter behaviour depends on what you're doing and the environment you're working in. You may need to create fake dependencies, or make use of mocking libraries. You might customise factories or roll your own dependency injection framework. Whichever strategy you choose, you'll need to write your code so that it is testable. One of the huge benefits of TDD is that the tests come first, guaranteeing that your code is testable.

- Testability needs to be designed in
  - TDD ensures code is testable
- Code with hidden dependencies is hard to test
  - Dependency Injection/Inversion
    Pass dependencies into code under test
    Write factories that permit injection of test doubles
  - Interfaces should be cohesive
    Wide interfaces encourage unnecessary coupling
  - Avoid globals, singletons etc.
- Retro-fitting unit tests is hard
  - Take small steps
  - Introduce a 'seam' – c.f. *Working Effectively with Legacy Code*

Necessity

I've previously said that we should only write code in response to a failing test, but that doesn't mean that every **method** has a test. Everything needs to be tested, but not every test is necessary.

- Test observable behaviour
  - Don't modify encapsulation to aid testing
  - If a behaviour isn't observable through public interface, what is it for?
- Don't slavishly write one test per method
  - Test behaviours
  - Some methods may not need any dedicated tests
  - Complex behaviours are likely to need many tests
- Choose test variants carefully
  - Edge conditions
  - Invalid inputs
  - Multiple invocations
  - Assert invariants
  - Error signalling

Granularity

There's plenty of advice about keeping methods small and classes focused. The same advice should be applied to unit tests. Each test should only exercise a SINGLE observable behaviour.

- It is tempting to combine related behaviours in a single test – DON'T… even if EXACTLY the same steps are needed

Understandability

Tests will fail. When they do you want to be able to fix the problem fast, so it's important that the tests themselves are understandable.

- Name tests to describe the behaviour under test
  - Describe nature of the test
    Is it checking that preconditions are enforced?
    Is a dependency going to signal an error?
  - Long names are fine – you only type them once
  - Be precise
    shouldReturnCorrectValue is not a good name for a test
    shouldReturnCorrectSumOfTwoIntegersWithoutOverflow
    should_return_correct_sum_of_two_integers_without_overflow
- When a test fails you want to know WHAT WENT WRONG
  - You don't want to reverse engineer the test
  - You don't want to run smaller tests to isolate the failure

Maintainability

Code evolves and the tests evolve with it. In my experience the maintainability of the test suite is the biggest hurdle to getting value from your unit tests.

- Unit Tests should be written to same quality as Production code
  - Tests will be maintained and read just as often as production code
  - Code is communication to other developers not just a compiler
- Organise tests into cohesive suites
- Refactor tests to avoid duplication
  - Use suites to perform common set up/tear down operations
  - Extract common code into methods
  - Extract common functionality into classes

As Martin Fowler [17] once said:

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

Cast iron rule

And let me be absolutely clear about the one CAST IRON RULE of unit tests:

No Unit Test should EVER depend on the outcome of any other Unit Test

Conclusion

Software developers are in the business of delivering value for their customers. I hope I've shown that tests, no matter what layer they are written at, can form the basis of an executable specification that can be used by:

- business people to validate that what they expected has been delivered
- developers to validate that what they have written does what they intended
- developers to validate that enhancements don't break existing functionality
- support staff to understand what the code was intended to do
- maintainers to ensure that defects get fixed and stay fixed

So, although this doesn't exactly answer the question I posed at the start, it does offer a way of allowing your customers to be involved in defining the criteria by which project success can be measured.

Finale

I've written this article top-down or outside-in. I couldn't see a way to develop it one feature at a time – there was no walking skeleton.

The literary diversions that I intended to include have been left as scribbles on small bits of paper. Consider yourselves lucky! ■

On a Game of Blockade

A student performs his analysis.

Recall that the Baron's game is comprised of taking turns to place dominos on a six by six grid of squares with each domino covering a pair of squares. At no turn was a player allowed to place a domino such that it created an oddly-numbered region of empty squares and Sir R----- was to be victorious if, at the end of play, the lines running between the ranks and files of the board were each and every one straddled by at least one domino.

When the Baron described his game to me it immediately struck me that this game was a splendid example of the pigeonhole principle!

The pigeonhole principle states that if we have *n* pigeonholes and *m* pigeons and that *m* is strictly greater than *n* then at least one pigeonhole must house more than one pigeon.

That mathematicians have elected to give a specific, and admittedly rather whimsical, name to this most blatantly obvious of observations might suggest that they are somewhat slow witted fellows, but the truly remarkable fact is that it is astonishingly useful for analysing games of this ilk.

Indeed, I expressed as much to the Baron, but he seemed somewhat distracted and I was disinclined from pressing the issue.

Now, given that the game concludes with the entire board covered in dominoes it is plain that it is impossible for a line to be straddled by just one domino; each is an even number of squares in length and must consequently be straddled by at least two.

Given that there are five lines running between the ranks and five between the files we must therefore use twenty dominoes to blockade them all. But there is but room enough on the board for eighteen and it is consequently impossible for Sir R----- to emerge victorious; indeed even if the Baron were playing to lose he should not have contrived to do so!

I should therefore certainly have advised Sir R----- to have declined the Baron's wager.

Now if we were to extend the game to the entire chessboard then this argument is rendered impotent; there is ample room for the requisite twenty eight dominos on a sixty four square board. This is no way demonstrates that it is actually possible to blockade all of the lines, however, and I must confess that my fellow students and I are at something of a loss as to whether it is possible to cover a full chessboard with dominoes such that they are. ■

cqf.com

CERTIFICATE IN QUANTITATIVE FINANCE CQF

Expand Your Mind and Career

Designed by quant expert Dr Paul Wilmott, the CQF is a practical six month-part time course that covers every gamut of quantitative finance, including derivatives, development, quantitative trading and risk management.

Find out more at **cqf.com**.

ENGINEERED FOR THE FINANCIAL MARKETS

Testing Times (continued)

References

[1] http://en.wikipedia.org/wiki/Do-Re-Mi

[2] *Growing Object Oriented Software, Guided by Tests*, Steve Freeman & Nat Pryce, Addison Wesley, 0321503627

[3] from a Nat Pryce/Steve Freeman presentation with permission

[4] *Crystal Clear: A Human-Powered Methodology for Small Teams*, Alistair Cockburn, Addison-Wesley, 0201699478

[5] http://fit.c2.com/

[6] http://fitnesse.org/

[7] http://cukes.info/

[8] *Bridging the communications gap*, Gojko Adzic, Neuri, 9780955683619

[9] http://testobsessed.com/2011/02/25/the-atdd-arch/

[10] http://www.xprogramming.com/Practices/PracNotNeed.html

[11] http://cleancoder.posterous.com/the-transformation-priority-premise

[12] http://www.infoq.com/interviews/coplien-martin-tdd

[13] http://www.pvv.org/~oma/TDDinC_Smidig2007.pdf

[14] http://www.stickyminds.com/s.asp?F=S13833_ART_2

[15] http://www.artima.com/weblogs/viewpost.jsp?thread=126923

[16] *Working Effectively with Legacy Code*, Michael Feathers, Prentice Hall, 0131177052

[17] *Refactoring: Improving the Design of Existing Code*, Martin Fowler *et al.*, Addison-Wesley, 0201485672

Coding Standards for Software Correctness

Yechiel Kimchi divides and conquers.

Coding Standard documents in the software industry are aimed at improving code quality. They are just a small part of the SW development process, that targets the same goal – software quality. Here I notice that well known and acceptable SW development processes generally bypass the role of the individual programmer, and concentrate on the processes that surround coding. In addition, existing coding standards, that are made to guide the individual programmer toward creating code of higher quality, are missing parts which, in my view, are at least as important as the parts they contain. In this article I suggest to approach the organization of Coding Standard documents in a different way, claiming that the new style will greatly improve software quality – even when all other things are kept the same.

Introduction

Correctness is arguably the single most important facet of a software tool – at least from the engineering point of view as part of the tool's quality. Indeed, correctness may be interpreted in different ways: from the widest interpretation as a synonym for *fulfilling all the tool's specifications* which may include even some marketing goals like *cool interface*, to one of narrowest interpretations as a synonym for *fulfilling all functionality specifications*. Correctness also includes performance, when it matters.

It is this narrow interpretation that I will concentrate on – suggesting a methodology for addressing correctness issues in Software Engineering. Notwithstanding this narrow interpretation, we know that despite the importance of marketing issues (appealing to customers, time-to-market, user interface, and so on), once users discover that too much functionality is broken, the tool is doomed (unless the tool is supplied by a monopoly).

In between there are correctness issues that are related to other engineering aspects of software development like maintainability, testability, robustness, modifiability, and many more [1], and all of them affect the tool's quality. As will be shown, most of the latter issues can also be addressed by the suggested technique.

As software projects became more and more complex, various methods were developed in order to deal with achieving software quality. Examples are *tools*: analyzers, test-drivers, environments, bug trackers, . . . and there are others: SW life-cycle models, methodologies (like XP [2]), and processes (like CM [3]), to name a few. All the above (with a few tools exceptions) approach quality at team level and above, up to the organization level: the individual developer is just a participant in a multi-player game. Nevertheless, we know that most of the developer's contribution is at the personal level: each developer contributes her own code, even if frequently consulting with her peers. By analogy, *developers are much more like a team of chess players than a football team*. Yet, when the Personal Software Process (PSP) was published [4], it too emphasized processes (as the term suggests) and did not directly address the actual mode of work of the individual. This may be understandable, given the fact that programming paradigms differ a lot in their practices. What was left is a set of (very good) recommendations like: log and measure your work, find your weaknesses, monitor your improvements, and so on.

YECHIEL KIMCHI

Yechiel Kimchi has a PhD in maths, has combined SW development and teaching since 1991, and bashes bad books on C and C++. He is interested in developing software that is correct, maintainable, and efficient. Contact him at yechiel.kimchi@gmail.com.

I will summarize this discussion with a quote of Bjarne Stroustrup[5], that I have found long after starting this project:

> Computer science must be at the center of software systems development. If it is not, we must rely on individual experience and rules of thumb, ending up with less capable, less reliable systems, developed and maintained at unnecessarily high cost. We need changes in education to allow for improvements of industrial practice.

Although producing high-quality code is, basically, the result of knowledge acquired through study and experience, many of the good practices can be summarized into a short style-guide doc-ument - and this is the subject of this article.

The article is organized as follows:

- 'Code quality – correctness' states the method I want to employ in verifying code correctness.
- 'Coding standard documents' describes current practices related to coding standard documents.
- 'Coding style for correctness' lays out the details of the suggested coding standard.
- Conclusions and future work.

Acknowledgment: A condensed, even terse, version of this article has appeared in the book *97 Things Every Programmer Should Know* [6].

Code quality – correctness

How do we check the correctness of the code in a given module, package, or even the whole project? If we had a formal tool, we could put the code with its specifications at one end, and get 'Yes' or a list of violations at the other end. On a smaller scale this is done in the hardware industry: With all due respect to millions of transistors in a CPU, SW projects have higher complexity in both control and number of states, and, therefore, automatic tools are still in their infancy. Thus, we end up with two main resources for verifying code quality:

- Developers' knowledge and attention at work.
- Testing: unit tests and QA teams.

As I mentioned in the introduction, no SW development process, model, or document explicitly says what a competent SW developer should do in order to elevate correctness of the code – except generally referring to the developer's knowledge and skills. Assuming the developer is indeed competent is not enough to be sure there are no errors, since we are all doomed to err from time to time: 'even experienced programmers inject a defect in every seven to ten lines of code' [7]. Since mechanical formal verification is still something to desire, and since testing usually tests the more common scenarios, we are left with manual formal verification – arguing that a piece of code does live up to its specifications: Probably the only way to verify that we do indeed check all scenarios, which is more than coverage. The bad news is that as the code grows beyond a few dozen lines (or when we try to be too formal), the proof that is required becomes longer and more error prone than the code. The good news is that we can skip the formal part of the proof, and argue about the code semi-formally; namely, we verify the code just as mathematicians verify their proofs. In essence, it does not take a big leap in order to claim that the code in front of us is the developer's **proof** that *the given specifications can be implemented*.

So here is the technique for verifying a given piece of code of whatever size. Although it is 'borrowed' from mathematics, it is already known in computer science – just not very often used:

1. The underlying approach is to divide all the code under consideration into short sections – from a single line, such as a function call, to blocks of less than ten lines – and arguing about their correctness. The arguments need only be strong enough to convince your devil's advocate peer developer.
2. A section should be chosen so that at each endpoint the state of the program (namely, the program counter and the values of all 'living' objects) satisfies an easily described property, and that the functionality of that section (state transformation) is easy to describe as a single task – these will make reasoning simpler.
3. The task is to argue that given the property at the beginning of a section, the code inside the section brings the program to a state with the property asserted at the section's end point.
4. Arguing about a series of sections is made by realizing that the state of the program at the end of one section is the state at the beginning of the next section.
5. Such endpoint properties generalize well known concepts like *precondition* and *post-condition* for functions, and *invariant* for loops, classes (with respect to their instances) and recursive functions.
6. Striving for sections to be as independent of one another as possible simplifies reasoning and is a treasure when these sections are to be modified.

Not only does the above sound simple enough, but it may be argued that developers already do it in a less intentional and less formal way than is described here. I agree, with one reservation: Look at your favorite bug-tracking system and count how many bugs could be prevented with a bit of additional care (like *off-by-one* loop errors). Indeed, they come mostly from thinking about the code in a casual way, which more often than not means oversight of the less probable paths.

Does the above guarantee bug-free code? Definitely not – but it does reduce the bug count dramatically. The coding standard that I am going to present is aimed at making this style of verification easy to handle – as easy as can be – given the complexities involved in program structure.

Coding standard documents

There is no doubt that coding standard documents are made for improving the production process of software tools/projects, and many goals are covered by the term *improving production process*. Some of the goals may be kind of coincidental – like having the code ready to be processed by a specific tool. (This is seen a lot in the HW industry, where verification tools have to read the textual description of the design.) However, in most cases the main reason is to elevate the quality of the code: the more safety issues are at stake, the more detailed and strict the standard is.

Standards usually begin by requiring some adherence to code layout and naming conventions: these make it easier for developers to become familiar with code developed by others, including checking its correctness and safely modifying it. Then, standards go on and require/encourage the use of safe constructs in the given programming language, and forbid/discourage the usage of constructs that are more error prone.

My observation is that there are two kinds of coding standards: short ones and long ones. I believe this characterization is serious, because from my personal experience the length of a document is highly correlated with its contents:

- Short documents tend to emphasize code layout, which is very important, but is just one factor of quality. They also cover some semantic issues, but not so many.
- Long documents tend to be very explicit in usage of the target programming language: very meticulous about do's and don'ts with certain constructs of it (the ones I know of are for safety critical products).

To summarize, existing short standards are manageable, but only partially cover what a coding guideline document should cover (virtually all documents I had to follow were short, concentrating on layout, and on the semantic part gave more don'ts than do's). On the other hand, existing long standards are much more detailed and close to being thorough in covering many aspects of quality: For example, JSF++ [8] is 140 pages, of which 50 pages just state the 221 rules with explanations, but without examples.

The shortcomings of the long documents, in my view, are two-folds: the first one is that a long document is harder to master. The second one is that the long documents tend to teach the developer about the target language. While it is not entirely redundant to recall some of the programming language properties (say, the lesser used), we should expect that those developers with less than a good working knowledge of the language have other resources to improve their technical knowledge (including in-house courses). For example, is the following (Rule #82 in [8]) reasonable: 'An assignment operator shall return a reference to `*this`.'

there are two kinds of coding standards: short ones and long ones

It should be noticed, however, that most developers that come with a good working knowledge of a language, do not necessarily have the knowledge – let alone the habit – of practising programming styles that emphasize correct code as the outcome. The reasons are out of the scope of this article.

There is another issue with many coding standards: they are only partially followed when not enforced by tools – and tools are not always available. It is my view that one of the main reasons coding standard are not very carefully followed is that many of them are perceived as stylistic, and not as introducing an important technical value.

Here I state the abstract concept that my suggested rules are based upon. The main contribution of this article is that they all can be argued to support easier verification of code correctness. The particular rules will be listed in the next section.

1. General coding-style rules are suggested. They are general enough not to be limited to a single programming language, though they are a better match for structured and object-oriented languages.
2. The rules are short and easy to remember, and being general they do not address a specific language, nor do they try to teach a specific language.
3. They all can be put under the title *make your code correct* with subdivisions of different flavours. Assuming that developers really want their code to be correct, they will have a tendency to better follow them, compared to following rules that sometimes seem to be arbitrary.
4. One subtle advantage of the rules to be presented is that once the developer understands them and their reasons, the developer can create more rules like them either by generalizations or by specializations, and above all **these rules need not be memorized** because they are directly based on the concept of programming.
5. Code layout rules, that always have some arbitrary flavour, are mentioned here but not stated – leaving it to be decided at company/project/group level.

Coding style for correctness

The rules to be presented here are organized in subsections, grouping together rules with similar flavour. All rules, however, have common characteristics: They are short, their justifications are short, and they all make the code easy to understand and argue about. And one last general comment: *Every rule has exceptions* but an exception should be a *well thought of* ***exceptional*** *case*.

Code layout

No coding style document ignores *code layout* issues. Indeed, this is the first thing one notices about a piece of code when first seeing it. It is also one of the topics that introduces flames in discussion lists. Code layout is transparent to the compiler but very important to the human being: an unfamiliar structure is much harder to decipher. Although I expect layout rules to be reasonable, the only important thing about them is that they are followed consistently. One can adapt to less reasonable rules, but can never adapt to varying/no rules. The more energy one invests in understanding the structure of the code, the less energy one has to argue about its correctness. Code layout begins with usage and locations of various parentheses, indentations, space around operators, alignment rules, and empty lines. Code layout rules refer to those practices that have no effect on the semantics of the program (e.g., indentation affects the semantics in Python, make, and most likely more), which can be controlled by tools (*beautifiers*, which make life easy). Comments do belong here (where, what, how and how much to comment).

Naming conventions are put here, but they are exceptional: there are no tools to enforce them, and they may have effect on the build process. (Some environments require that names are distinguished by a short prefix, even as short as six characters.) With well-chosen names, the need for comments may be dramatically reduced. The basic rule is: Choose descriptive, but not too long names. Tell as much as possible using names (variables, types, functions), and ‘Comment Only What the Code Cannot Say’ (Kevlin Henney, in [6]).

Some environments require that names are distinguished by a short prefix, even as short as six characters

I’ll conclude this section with a personal observation about code layout: Over the years I have reviewed, browsed and leafed through many C and C++ books. Some of my comments of the bad ones were recorded on a dedicated Internet page at the Technion (Israel) [9] (BadBooks). It is my strong personal impression that there is a very high correlation between the code layout of a book and its technical correctness and value.

Section independence

Our desire is to have each code section (a verification atom) as independent of each other as possible, since interactions among sections are harder to argue about. This justifies the following rules:

- No usage of `goto`s: a `goto` explicitly makes remote sections highly interdependent.
- No modifiable global variables: they silently make remote sections highly dependent. More explicitly, when a function uses a global variable, one cannot argue that ‘the function is correct’ because each call to the function makes a different challenge to argue about. You add a new call for that function? Start arguing its correctness all over again.
- Limit the scope of variables to the absolute minimum: If you don’t, you’ll repeatedly argue *this variable is not used in this section* – why bother? You’ll also avoid unintentional usage of it.
- Make objects *immutable* whenever relevant: some object must be accessed by several sections, but only a few will modify them. Make such an object appear immutable to other sections, and you’ll be able to go home earlier.

Simplify sections

The simpler a section is, the shorter the argument for its correctness. (Note that empty sections have no bugs :-) but missing ones *are* bugs.) Usually a section is a function body or a block of code (e.g., a loop body).

- Make your sections short and doing a single task.
- If you perceive a nested section, factor it out as a function.
- Avoid deeply nested blocks, not just by extracting some as functions, but also by restructuring them (see ‘Spartan Programming’ [10] for some techniques).
- Functions should be short: recommended upper bound is 30 lines. Recommended size – less than 20 lines. No matter how modern is our screen – its size and its fine resolution – our brain hasn’t changed for the last 20 or maybe 50 thousand years: it has a limited capacity.

A note on long functions: In the 1960s, the practice was to limit functions to < 24 lines. One may attribute it to the number of lines of text screens at that time, but this is a very partial perception of the situation: First, most of debugging was done when reading a printout (and typing was done by punching cards) without this limitation. Second, if manufacturing a monitor was so expensive at that era, why monitors were not built to have 15 lines of text? There must have been some considerations of cost-effectiveness. In practice, I rarely needed functions longer than 15 lines, and seeing functions longer than 50 LOC is not just a nuisance, but also reveals low level of abstraction as well as being an obstacle in achieving a higher one. Incidentally, the longest function I’ve actually seen was 555 LOC, and all I can say is that in contrast to car engines, the bigger your function is, the weaker it is.

Narrow interfaces

Sections cannot be completely independent, but after minimizing the number of sections that communicate with a given section, we would benefit if the communication between any pair is limited to the minimal necessary and it is in its simplest form. This is the sole purpose of *encapsulation*. The developer should remember that more communication has a detrimental effect on provability, since each channel of communication and each data that passes through such a channel should be argued and be proven correct. In practice we don’t think about sections that communicate, but rather about functions, objects and modules that communicate. Note that minimizing the scope of a variable is a means to limit communication between a nested block and its surroundings. However, some services are inherently communication intensive, and this issue is dealt with in ‘Stable states’ (below).

- Functions should have only a few parameters (four is a good upper-bound). If the initial version has too many parameters, try to aggregate some of them, which are interrelated, into a single object with designated states (as in ‘Stable states’, below). If this is not enough, maybe the function has too many responsibilities – break it to smaller functions.
- Minimize exposure of data items: They should be wrapped by a function, or by an object, or by other means of interface (and be *private*). Limiting the usage of an object by discipline is nice, but limiting usage by a restrictive interface is trustworthy.
- The interface of an object (say, via its class) should be *minimal and complete* – and *complete* is context dependent: just to support what is reasonably required, and likewise for modules.
- The usage of *getters* (interface constructs that expose the internal state) should be discouraged: Not only does this break the rule

  Don’t ask an entity for data to do your job, tell it to do the job for you with the data it already has.

  but it makes the surrounding code dependent on the **type** of that data – which may prevent future modifications of the code.

Stable states

Some entities are not useful without a wide and rich interface. A simple example may be containers (C++’s STL containers is an example). If they had a narrow interface there would be the need for many similar containers (e.g., lists) that differ only by the services they provide – which is ridiculous. So we’d rather have a single service (a class) that is usable for different situations thanks to its wide interface. Here we manage complexity internally, rather than externally. The key concept is *invariant* – a property (set of properties) that is **always true** for that entity (be it an object, a module or the state of the program at the end of a loop body). When we design such an entity, we define a set of properties that will guarantee that the entity is still usable by its clients. Then we implement the entity so that each of its zillion interface functions guarantees that it preserves all those properties when it operates with the entity – an initialization process is a must in such a case. This collection of properties is called *the **invariant** of the entity*. In such a case we cannot argue that such an entity will always give the correct results for the client, but we can prove that if the client communicates correctly with that entity, that entity will provide the expected results. Simple objects do not usually have invariants: any state of an `int` in C is legal. On the other hand, if you want an `int` object that can only be incremented, it's not enough to wrap it with functions that only increment its value – you should also make sure that it doesn't overflow.

- Every entity that is not trivial should be designed with an invariant in mind, and its interface should preserve it (an entity may be a function, if it has an internal state).

Immediate consequences of the above are:

- Avoid using *setters* (interface constructs that are made to modify an attribute, usually by modifying a single data member). because setters tend to break class invariants.
- Modifiers should be expressed as giving services to clients – not as receiving orders from them – because the client is oblivious to how a request is fulfilled.

Performance

Performance issues are delicate. On the one hand, here is my version of an old rule: *make your software right, then better, then fast*. On top of that there is reality that shows that performance considerations are very sensitive to environment: the language that is used, the compiler that is used and its exact version, and on top of all that there are hardware and communication issues – all of those impact performance more than the developer can usually change. Nevertheless, there are a few rules that are not hard to follow – and more importantly, do not make the software more complicated. Such rules will have positive contribution to performance under all circumstances (where a language supports that facility). Here are some of them:

- Use well established algorithms and data structures.
- Not all algorithms scale up well (e.g., matrix multiplication) and some algorithms do not scale down well (e.g., binary search over an array is worse than linear search for short arrays – the sweet point is around 20 elements).
- Objects that are not modified should be annotated as `const`. The compiler may avoid rereading them.
- Use operators that minimize creation of temporary objects.
- Factor out of loops values that are not modified by the loop, especially calculations among them.
- Help the compiler inlining functions: make your functions short. (A language may implicitly support inlining, cf. pre-C99 C.)
- Allow the compiler to utilize the cache: make functions and loop blocks short – very short.
- When branching, put the main branch first – it may save loading the other branches. However, there may be other considerations, like making the *shorter* branch the first one – especially if it's an error state with a return, that allows elimination nesting of the `else` block.

These are relatively simple rules that are either abstract or invoke the hardware but are language independent. Note that some of them already appeared before, as providing easier to verify structures.

Conclusions

Integrating the outcome of 'Coding style for correctness' results in software that is built of loosely coupled relatively small components, which are known to improve many desirable SW characteristics, like maintainability, robustness, modifiability, testability and more. In this work I have presented a set of coding rules that are relevant to quite a few common programming languages. All these rules were chosen to help arguing about software correctness – though many of them are perceived as stylistic only. However, each specific language should have more rules adequate for its own idiosyncrasies. For example, I won't consider many rules for C++ that deal with *undefined* behaviour in the language (this is the responsibility of the developer to know the language), with maybe exceptions for the more obscure cases. But I may add rules related to ownership of objects, although in most cases this is solved now by `shared_ptr`.

I have not addressed concurrent programming, but considerations there are basically the same, while the required argumentation is much more complicated.

There are two non-conflicting directions to continue from here: One is to describe more useful rules, and the other is to design a tool that will assist developers to comply with these rules. ■

References and further reading

[1] Abran A. and Moore J., *SWEBOK, Guide to the Software Engineering Body of Knowledge (Ironman)*, IEEE Computer Society, 2004

[2] Beck K., *Extreme Programming Explained: Embrace Change*, Addison-Wesley,1st ed. 1999, 2nd ed. 2004

[3] CMMI Product Team, *CMMI for Development (Ver. 1.2)*, Carnegie Mellon, Software Engineering Inst., 2006

[4] Humphrey W., *A Discipline for Software Engineering* (1st ed.), Addison-Wesley Longman, 1995

[5] Stroustrup B. 'What should we teach new software developers? Why?' *Communications of the ACM*, v.53, #1 (2010), pp. 40–42

[6] Henney K. (ed.), *97 Things Every Programmer Should Know*, O'Reilly, 2010, 30–31. http://programmer.97things.oreilly.com/wiki/index.php/Coding with Reason

[7] In Humphrey W., 'SEI-2000-TR-022' http://www.sei.cmu.edu/reports/00tr022.pdf from Hayes W. and Over J. W., 'Technical Report, CMU/SEI-97-TR-001', http://www.sei.cmu.edu/reports/97tr001.pdf, 1997

[8] Doc.# 2RDU00001 Rev C, *Joint Strike Fighter Air Vehicle C++ Coding Standards*, Lockheed Martin Corp., 2005

[9] Kimchi Y., *C and C++ Bad Books*, CS Dept. The Technion, Israel http://www.cs.technion.ac.il/users/yechiel/CS/BadBooksC+C++.html, 1998-2007

[10] Gil Y. 'Spartan Programming', CS Dept. The Technion, Israel http://ssdl-wiki.cs.technion.ac.il/wiki/index.php/Spartan programming 1996-2011

[11] Kernighan B. and Ritchie D., *The C Programming Language* 2nd ed., Prentice Hall, 1988

while (you care about code)
{
read (cvu && overload);
}
do(it);
because good code matters
ACCU
www.accu.org
PROFESSIONALISM IN PROGRAMMING

Concurrent Programming in Go

Alexander Demin examines Google's Go language.

Concurrent programming, multi-threading, scaling... These trends are becoming more and more popular even for desktop computing. And the ability to write such applications is a must for every professional programmer.

Almost all modern programming languages support multi-threading, but not every language allows you to do it easily and safely. There are a few emerging languages on the market with advanced multi-threading capabilities, and one of them is Go.

Go was designed at Google to make handling concurrency easy. In this article I will try to inspire some interest in Go, to give a taste of this very interesting language for those who haven't yet had time to look at it.

I will use a small but quite functional program as an example which I wrote recently in Go to automate some parts of regression testing at the company where I work.

This program does a few quite useful routine things: it deals with a TAR archive, parses a command line and also does advanced multi-threaded business logic. I hope you'll get an initial overview of Go and maybe a vision of how it could be applicable in your area of business.

We have a compiler for a proprietary programming language translating a dialect of BASIC into C.

For historical reasons we have no well defined regression testing for this compiler. Recently we decided to improve it and create extensive testing based on sources of a business application written in that BASIC.

The plan is to freeze a particular release of the compiler having no open issues, then to compile a significant amount of code and also freeze the results. Those results will be a master copy, and will be used for comparison against the output produced by a modified version of the compiler in the future. It doesn't protect from bugs completely but at least it guarantees against breaking existing functionality.

It's pretty trivial issue, but we have a problem. There are more than fifteen thousand files, at about 1G total in size (for convenience, it is all packed into a TAR). Such a run could take a long time, and there is a straightforward requirement to speed it up using multi CPU hardware. The business logic of the task is easily parallelizeable.

To begin with, we can write a Makefile and simply run GNU Make with the `-j` flag [specifies the number of jobs to run simultaneously. Ed] . But a special, purely native program could do better.

Obviously, instead of running the compilation sequentially, we can launch each compilation in a separate thread. But with more than ~15K files it won't be efficient to start so many threads at the same time. It's much better to have a pool of threads, whose size is constant and depends on the number of CPUs. We will be assigning every task to a free thread, and if all the threads are occupied, we will be blocked until at least one of those gets released.

Thus, we will be maintaining *N* threads utilizing CPUs more optimally without unnecessary context switching.

I decided to attempt this myself in Go. You may not believe it but this program is only the second time I have used Go (the first one was a benchmark of Eratosthenes Sieve [1] amongst C, C++ and Go), and after an hour I had almost working code.

The program unpacks a given TAR archive and compiles every single file. The key here is to run each compilation in a separate thread, with the threads managed by a thread pool.

The main concept adopted in the code is Go channels. The channels are a synchronous mechanism of communication between threads. Instead of sharing memory and using a synchronization primitive such as a mutex, semaphore or conditional variable to manage concurrent access, the channels allow us to use another concept: sharing by communication. The information which needs to be shared or sync'ed amongst multiple threads is sent over the channels. This makes the concurrent code easier to read and debug because it can be logically isolated from other threads (primarily from their data). Data exchange with other concurrent execution flows is achieved by blocking both sender and receiver on the channel.

Channels are a language feature, provided by the Go runtime. All the complexity of the implementation is hidden from the application developer

Channels are a language feature, provided by the Go runtime. All the complexity of the implementation is hidden from the application developer.

The second key feature of Go which we will look at is Goroutines. Goroutines are similar to native OS threads but not exactly the same. From the Go documentation:

> Goroutines are multiplexed onto multiple OS threads so if one should block, such as while waiting for I/O, others continue to run. Their design hides many of the complexities of thread creation and management.

Goroutines are similar to Erlang lightweight processes. The number of Goroutines you can start is usually limited only by memory. Every Goroutine has its own small stack which makes them very cheap to spawn. The syntax of starting a Goroutine is made simple using the **`go`** keyword:

```
...
go BackroundTask()
...
```

Now we are ready to follow the code. The most interesting thing is that the call to **`compile()`** is the same here as it would be in a naive sequential approach (see Listing 1).

Makefile:

```
target = tar_extractor
all:
  6g $(target).go
  6l -o $(target) $(target).6
```

I tested this application on a 64-bit Linux blade with 8 CPUs. During testing I was the only user of that box, so the results are directly comparable. The `huge.tar` file contains ~15 thousand source files and its size is ~1GB.

ALEXANDER DEMIN

Alexander Demin is a software engineer with a PhD in Computer Science. Constantly exploring new technologies he is always ready to drill down into the code with a disassembler to prove that the bug is out there. He may be contacted at alexander@demin.ws.

Listing 1

```
package main
import (
      "archive/tar"
      "container/vector"
      "exec"
      "flag"
      "fmt"
      "io"
      "os"
      "strings"
)
// There are two flags: a number of threads and a
// compiler name.
var jobs *int = flag.Int("jobs", 0,
    "number of concurrent jobs")
var compiler *string = flag.String("cc",
    "bcom", "compiler name")
func main() {
  flag.Parse()
  os.Args = flag.Args()
  args := os.Args
  ar := args[0]
  r, err := os.Open(ar, os.O_RDONLY, 0666);
  if err != nil {
    fmt.Printf("unable to open TAR %s\n", ar)
    os.Exit(1)
  }
  // defer is similar to "finally {}", it
  // guarantees execution of the code when a
  //scope goes out.
  defer r.Close()
  // This is a TAR unpacking TAR.
  fmt.Printf("- extracting %s\n", ar)
  // Create a TAR reader.
  tr := tar.NewReader(r)
  tests := new(vector.StringVector)
  // Traverse the archive sequentially collecting
  // the file names to compiler later.
  for {
    // Get the next file descriptor from archive.
    hdr, _ := tr.Next()
    if hdr == nil {
      break
    }
    name := &hdr.Name
    // If it is not a header file, we keep
    // its name.
    if !strings.HasPrefix(*name, "HDR_") {
      tests.Push(*name)
    }
    // Create a new file.
    w, err := os.Open("data/" + *name,
       os.O_CREAT | os.O_RDWR, 0666)
    if err != nil {
      fmt.Printf("unable to create %s\n", *name)
        os.Exit(1)
    }
    // Copy the file content from the archive
    // to the disk.
    io.Copy(w, tr)
    w.Close()
  }
  fmt.Printf("- compiling...\n")
  *compiler , _ = exec.LookPath(*compiler)
  fmt.Printf("- compiler %s\n", *compiler)
  if *jobs == 0 {
    // Call "compile()" sequentially, in the
    // main thread.
```

Listing 1 (cont'd)

```
    fmt.Printf("- running sequentially\n")
    for i := 0; i < tests.Len(); i++ {
      compile(tests.At(i))
    }
  } else {
    // Call "compiler()" in parallel threads.
    fmt.Printf(
       "- running %d concurrent job(s)\n", *jobs)
    // This is a task channel, where we put the
    // names of the files we want to compile.
    // Runnner threads wait for messages on this
    // channel, which has limited capacity to do
    // throttling similar to semaphore. The empty
    // name means we are asking the runner thread
    // to wrapup.
    tasks := make(chan string, *jobs)
    // This is a runner thread wrapup acknowledge
    // channel. The main thread waits on the
    // channel until all the runners confirm
    // their wrapup. The type of messages
    // in this channel is not relevant here.
    done := make(chan bool)
    // Launch the runners.
    for i := 0; i < *jobs; i++ {
      go runner(tasks, done)
    }
    // Iterate through the list of files and pass
    // the names into the channel. If there are
    // no free runners this thread is blocked.
    for i := 0; i < tests.Len(); i++ {
      tasks <- tests.At(i)
    }
    // Send exit command to runners and wait for
    // confirmation from all of them.
    for i := 0; i < *jobs; i++ {
      tasks <- ""
      <- done
    }
  }
}
// Runner thread.
func runner(tasks chan string, done chan bool) {
  // Infinite loop.
  for {
    // Wait for a message. Usually the thread is
    // blocked here.
    name := <- tasks
    // If the name is empty, we wrapup.
    if len(name) == 0 {
      break
    }
    // Compile the file.
    compile(name)
  }
  // Send message to wrapup acknowledge channel.
  done <- true
}
func compile(name string) {
  // Call the compiler.
  c, err := exec.Run(*compiler, string{*compiler,
     name}, os.Environ(), "./data", exec.DevNull,
     exec.PassThrough, exec.PassThrough)
    if err != nil {
      fmt.Printf("unable to compile %s (%s)\n",
         name, err.String()) os.Exit(1)
    }
    c.Wait(0)
}
```

This is a report of the CPU load when the machine is doing nothing (all CPUs are almost 100% idle):

CPU	us	sy	ni	id	wa	hi	si	st
0	0.0%	0.0%	0.0%	100.0%	0.0%	0.0%	0.0%	0.0%
1	0.0%	0.0%	0.0%	99.7%	0.3%	0.0%	0.0%	0.0%
2	0.0%	0.0%	0.0%	100.0%	0.0%	0.0%	0.0%	0.0%
3	0.0%	0.0%	0.0%	100.0%	0.0%	0.0%	0.0%	0.0%
4	0.0%	0.0%	0.0%	100.0%	0.0%	0.0%	0.0%	0.0%
5	0.0%	0.3%	0.0%	99.3%	0.3%	0.0%	0.0%	0.0%
6	0.0%	0.0%	0.0%	100.0%	0.0%	0.0%	0.0%	0.0%
7	0.0%	0.0%	0.0%	100.0%	0.0%	0.0%	0.0%	0.0%

Run it in a sequential mode (`-jobs 0`):

```
make && time -p ./tar_extractor -jobs 0 huge.tar
```

Time:

real	213.81
user	187.32
sys	61.33

Almost all the CPUs are 70–80% idle (I did the snapshots during the compilation phase):

CPU	us	sy	ni	id	wa	hi	si	st
0	11.9%	4.3%	0.0%	82.5%	1.3%	0.0%	0.0%	0.0%
1	9.6%	2.7%	0.0%	87.7%	0.0%	0.0%	0.0%	0.0%
2	4.3%	1.3%	0.0%	92.7%	1.7%	0.0%	0.0%	0.0%
3	16.0%	6.0%	0.0%	78.0%	0.0%	0.0%	0.0%	0.0%
4	12.6%	4.3%	0.0%	82.7%	0.3%	0.0%	0.0%	0.0%
5	11.6%	3.3%	0.0%	85.0%	0.0%	0.0%	0.0%	0.0%
6	4.7%	1.3%	0.0%	94.0%	0.0%	0.0%	0.0%	0.0%
7	16.6%	6.3%	0.0%	77.1%	0.0%	0.0%	0.0%	0.0%

The total CPU load is 2.7% (Figure 1).

Now run with the thread pool but using one runner only (`-jobs 1`).

Time:

real	217.87
user	191.42
sys	62.53

CPUs:

CPU	us	sy	ni	id	wa	hi	si	st
0	5.7%	1.7%	0.0%	92.7%	1.3%	0.0%	0.0%	0.0%
1	9.6%	5.3%	0.0%	81.3%	0.0%	0.0%	0.0%	0.0%
2	7.0%	2.7%	0.0%	89.3%	1.3%	0.0%	0.3%	0.0%
3	15.3%	5.7%	0.0%	77.7%	0.0%	0.0%	0.0%	0.0%
4	6.0%	2.0%	0.0%	92.0%	0.0%	0.0%	0.0%	0.0%
5	14.3%	7.3%	0.0%	78.4%	0.0%	0.0%	0.0%	0.0%
6	7.0%	2.3%	0.0%	90.7%	0.0%	0.0%	0.0%	0.0%
7	15.3%	6.6%	0.0%	78.1%	0.0%	0.0%	0.0%	0.0%

Obviously, it is all similar because we are running only one thread.

Now run with multiple runners in the pool (`-jobs 32`):

```
make && time -p ./tar_extractor -jobs 32 huge.tar
```

Time has been reduced up to 7 times.

The overall CPU load is now 23% (again measurements taken during the compilation phase) – see Figure 2.

In fact, all the CPUs are quite busy:

CPU	us	sy	ni	id	wa	hi	si	st
0	56.3%	26.3%	0.0%	17.3%	0.0%	0.0%	0.0%	0.0%
1	55.5%	27.9%	0.0%	15.6%	1.0%	0.0%	0.0%	0.0%
2	56.1%	25.9%	0.0%	15.0%	0.7%	0.3%	2.0%	0.0%
3	58.1%	26.2%	0.0%	15.6%	0.0%	0.0%	0.0%	0.0%
4	57.2%	25.8%	0.0%	17.1%	0.0%	0.0%	0.0%	0.0%
5	56.8%	26.2%	0.0%	16.9%	0.0%	0.0%	0.0%	0.0%
6	59.0%	26.3%	0.0%	13.0%	1.7%	0.0%	0.0%	0.0%
7	56.5%	27.2%	0.0%	16.3%	0.0%	0.0%	0.0%	0.0%

The point of this testing was to demonstrate how even the simplest concurrent code speeds everything up several times. We can also see how it is easy and relatively safe to write multi-threaded imperative code in Go.

Going further, I decided to compare the performance of Goroutines and Boost (C++) Threads. Basically, this is a competition between pthreads and Goroutines. I agree that such a comparison is not quite fair because we are comparing native OS threads wrapped by Boost with Goroutines, which are lightweight threads multiplexed onto OS threads. However, at the same time we are comparing the APIs facing a developer, and this gives us a feel for which API is nicer to use as well as benefiting from mult-threading.

I agree that using the queue with a lock is not efficient here and can be replaced with a lock-free implementation. But, again, this question is about which API is easier for concurrent programming: in Go you don't need to remember about using lock-free containers whenever possible (you may end up with a situation when a container cannot be implemented lock-free at all), you solve your business problem instead of dealing with implementation details.

I've created a simple program in both languages. This program needs to execute many requests of the same type. The payload of the request will be a little, almost empty, function. This will allow the thread management to be stressed rather than comparing the speed of the generated code.

Listing 2 shows the code in Go, and the Makefile to build and run it with different parameters is in Listing 3.

Listing 4 shows the code in C++, and the Makefile for running the tests is in Listing 5.

In both programs we will be using 8 CPUs and the number of tasks will vary.

Basically, this is a competition between pthreads and Goroutines

Figure 1

```
PID USER       PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
15054 tester   18   0 41420 4980 1068 S  2.7  0.1   0:02.96 tar_extractor
```

Figure 2

```
PID USER       PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
17488 tester   16   0 45900 9732 1076 S 23.6  0.1   0:06.40 tar_extractor
```

Listing 2

```
package main

import (
        "flag"
        "fmt"
)

var jobs *int = flag.Int("jobs", 8,
   "number of concurrent jobs")
var n *int = flag.Int("tasks", 1000000,
   "number of tasks")

func main() {
  flag.Parse()
  fmt.Printf("- running %d concurrent job(s)\n",
     *jobs)
  fmt.Printf("- running %d tasks\n", *n)
  tasks := make(chan int, *jobs)
  done := make(chan bool)
  for i := 0; i < *jobs; i++ {
    go runner(tasks, done)
  }
  for i := 1; i <= *n; i++ {
    tasks <- i
  }
  for i := 0; i < *jobs; i++ {
    tasks <- 0
    <- done
  }
}

func runner(tasks chan int, done chan bool) {
  for {
    if arg := <- tasks; arg == 0 {
      break
    }
    worker()
  }
  done <- true
}

func worker() int {
  return 0
}
```

Listing 3

```
target = go_threading
all: build
build:
  6g $(target).go
  6l -o $(target) $(target).6

run:
  (time -p ./$(target) -tasks=$(args) \
     1>/dev/null) 2>&1 | head -1 | \
     awk '{ print $$2 }'

n = \
10000 \
100000 \
1000000 \
10000000 \
100000000

test:
  @for i in $(n); do \
  echo "`printf '% 10d' $$i`" \
  `$(MAKE) args=$$i run`; \
  done
```

Listing 4

```
#include <iostream>
#include <boost/thread.hpp>
#include <boost/bind.hpp>
#include <queue>
#include <string>
#include <sstream>
class thread_pool {
  typedef boost::function0<void> worker;
  boost::thread_group threads_;
  std::queue<worker> queue_;
  boost::mutex mutex_;
  boost::condition_variable cv_;
  bool done_;
  public:
  thread_pool() : done_(false) {
    for(int i = 0;
       i < boost::thread::hardware_concurrency();
       ++i)
      threads_.create_thread(boost::bind(
         &thread_pool::run, this));
}
void join() {
  threads_.join_all();
}
void run() {
  while (true) {
    worker job;
    {
      boost::mutex::scoped_lock lock(mutex_);
      while (queue_.empty() && !done_)
        cv_.wait(lock);
      if (queue_.empty() && done_) return;
      job = queue_.front();
      queue_.pop();
    }
    execute(job);
  }
}
void execute(const worker& job) { job(); }
void add(const worker& job) {
  boost::mutex::scoped_lock lock(mutex_);
  queue_.push(job);
  cv_.notify_one();
}
void finish() {
  boost::mutex::scoped_lock lock(mutex_);
  done_ = true;
  cv_.notify_all();
}
};
void task() {
volatile int r = 0;
}
int main(int argc, char* argv[]) {
  thread_pool pool;
  int n = argc > 1 ? std::atoi(argv[1]) : 10000;
  int threads =
     boost::thread::hardware_concurrency();
  std::cout << "- executing " << threads <<
     " concurrent job(s)" << std::endl;
  std::cout << "- running " << n << " tasks" <<
     std::endl;
  for (int i = 0; i < n; ++i) {
    pool.add(task);
  }
  pool.finish();
  pool.join();
  return 0;
}
```

Relish the Challenge

Pete Goodliffe encourages us to seek out a new challenge.

Success is not final, failure is not fatal: it is the courage to continue that counts

~ Winston Churchill

We are 'knowledge workers'. We employ our skill and knowledge of technology to make good things happen. Or to fix it when they don't. This is our joy. It's what we live for. We revel in the chance to build things, to solve problems, to work on new technologies, and to assemble pieces that complete interesting puzzles.

We're wired that way. We relish the challenge.

The engaged, active programmer is constantly looking for a new, exciting challenge.

Take a look at yourself now. Do you actively seek out new challenges in your programming life? Do you hunt for the novel problems, or for the things that you're really interested in? Or are you just coasting from one assignment to the next without much of a thought for what would motivate you?

Can you do anything about it?

The whys and wherefores

Working on something stimulating, something challenging, on something that you enjoy getting stuck into helps keep you motivated.

If you instead get stuck in the coding 'sausage factory', just churning out the same tired code on demand, you will stop paying attention. You will stop learning. You will stop caring and investing in crafting the best code you can. The quality of your work will suffer. And your passion will wane.

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

Concurrent Programming in Go (continued)

Starting up C++ code:

```
 make && make -s test
 g++ -O2 -I ~/opt/boost-1.46.1 -o boost_threading \
            -lpthread \
            -lboost_thread \
            -L ~/opt/boost-1.46.1/stage/lib \
            boost_threading.cpp
(time -p LD_LIBRARY_PATH= \
  ~/opt/boost-1.46.1/stage/lib ./boost_threading  \
  1>/dev/null) 2>&1 | head -1 | awk '{ print $2 }'
   10000 0.03
  100000 0.35
 1000000 3.43
10000000 29.57
100000000 327.37
```

Now run Go:

```
  make && make -s test
  6g go_threading.go
  6l -o go_threading go_threading.6
     10000 0.00
    100000 0.03
   1000000 0.35
  10000000 3.72
  100000000 38.27
```

The difference is obvious. Go code is ~9 times faster.

Finally, I dug a bit further. There is an environment variable in the Go runtime called **GOMAXPROCS** which defines how many CPUs the Go runtime is allowed to use. Surprisingly the default value is 1! It means in the exercise above the program in Go used only one CPU while the C++ one used 8. When I set **GOMAXPROCS** to 8 and repeated the test, the time of the Go version increased significantly and become almost the same as in C++. A possible conclusion is that scheduling threads via the OS scheduler (when native threads are involved) is more expensive than multiplexing lightweight threads onto one native thread.

To recap, I don't want to try to convince anybody that C++ is bad for writing concurrent programs, of course not. Moreover, the C++ code here can be significantly improved by using a lock-free queue. I just wanted to highlight that the API to deal with concurrency can be different, and what Go offers is definitely worth having a look at.

Happy Going...■

Note

[1]Eratosthenes Sieve: http://en.wikipedia.org/wiki/Sieve_of_Eratosthenes

Listing 5

```
BOOST = ~/opt/boost-1.46.1
target = boost_threading
build:
  g++ -O2 -I $(BOOST) -o $(target)
      -lpthread \
      -lboost_thread \
      -L $(BOOST)/stage/lib \
      $(target).cpp
run:
  (time -p LD_LIBRARY_PATH=$(BOOST)/stage/lib \
  ./$(target) $(args) \
  1>/dev/null) 2>&1 | head -1 | awk '{ print $$2
}'
n = \
10000 \
100000 \
1000000 \
10000000 \
100000000

test:
  @for i in $(n); do \
    echo "`printf '% 10d' $$i`" \
    `$(MAKE) args=$$i run`; \
    done
```

Conversely, actively working on coding problems that challenge you will encourage, excite you, and help you to learn and develop. It will stop you becoming staid and stale.

Nobody likes a stale programmer. Least of all, yourself.

What's the challenge?

So what is it that particularly interests you?

It might be that new language you've been reading about. Or it might be working on a different platform. It might just be trying out a new algorithm or library. Or to kick off a pet project. Or to attempt an optimisation or refactor of your current system, just because you think you see an improvement (even if – shudder to think – it doesn't provide business value).

Often this kind of personal challenge can only be gained on a side-project; something you work on alongside the more mundane day-to-day tasks. And that's perfect – it's the antidote to dull development. The programming panacea. The crap code cure.

What excites you as a programmer? Look at the tasks you are working on.

- Are you happy just to be paid for producing code?
- Do you want to be paid because you will produce a particularly exceptional job?
- Are you performing that task solely for the kudos; are plaudits are enough for you?
- Would the many open-source eyeballs following your project give you a deep sense of satisfaction?
- Do you want to be the first person to provide a solution in a new niche?
- Do you want to solve a problem just for the intellectual exercise in doing do?
- Is it just a project that particularly interests you, and your strange peccadilloes?
- Or is it an entrepreneurial project – something you think will one day make you millions?

As I look back over my career, I can see that I've worked on things in each of those camps. But I've had the most fun, and produced the best software, when working on projects that I've been invested in, where I've cared about the project itself, as well as wanting to write exceptional code.

Don't do it!

Of course, there are a potential downsides to seeking out cool coding problems for 'the fun of it':

- It's selfish to steer yourself towards exciting things all the time, leaving boring stuff for other programmers to pick up.
- It's dangerous to 'tinker' a working system just for the sake of the tweak, if it's not introducing real business value. You're adding unnecessary change and risk. And it's a waste of time that could be invested elsewhere more profitably.
- If you get side-tracked on pet projects or little 'science experiments', then you'll never get any 'real' work done.
- Remember: not every programming task is glamourous or exciting. A lot of our day-to-day tasks are mundane plumbing. That's just the nature of programming in the real world.
- Re-writing something that already exists is a gross waste of effort. You are not contributing to our profession's corpus of knowledge. You are likely to just recreate something that already exists, not as good as existing implementations, and full of new terrible bugs. What a waste of time!

Yada. Yada. Yada.

These positions are valid, to a point.

But this it is exactly because we have to preform dull tasks that we should also seem to balance them with the exciting challenges. We must be responsible in how we do this, and whether we use the resulting code.

Get challenged

So work out what you'd like to do. And do it.

- Perform some code katas that will provide valuable deliberate practice. Throw the code away afterwards.
- Find something you'd like to write code for, just for the fun of it.
- Kick off a personal project. Don't waste all your spare time on it, but find something you can invest effort in that will teach you something new.
- Maintain a broad field of personal interest, so you have good ideas of other things to investigate and learn from.
- Don't ignore other platforms and paradigms. Try re-writing something you know and love on another platform or in another kind of programming language. Compare and contrast the outcome. Which environment lent itself better to that kind of problem?
- Consider moving on if you're not being stretched and challenged where you're currently working. Don't blindly accept the status quo! Sometimes the boat needs to be rocked.
- Work with, or meet up with other motivated programmers. The recent ACCU conference was an especially good example of this. Attendees some back with a head full of new ideas, and invigorated from the enthusiasm of their peers.
- Make sure you can see the progress you're making. Review source control logs to see what you've achieved. Keep a daily log, or a TODO list. Enjoy knocking off items and making headway.
- Keep it fresh: take breaks so you don't get overwhelmed, stifled, or bored by bits of code.
- Don't be afraid of reinventing the wheel! Write something that has already been done before. There is no harm in trying to write your own linked-list, or standard GUI component. It's a really good exercise to see how yours compares to existing ones. (Just be careful how you employ them in practice.)

Conclusion

Yes, I've been hand waving. And I've slipped into motivational speaker territory (again). But this stuff is important. Do you have something you're engaged in and love to work on?

It's impractical and dangerous to just chase shinny new things all the time and not write practical, useful code. But it's also personally dangerous to get stuck in a coding rut, only ever working on meaningless, tedious software. ■

Questions

1. Do you have projects that challenge you and stretch your skills?
2. Are there some ideas you've thought about for a while, but not started? Why not start a little side-project?
3. Do you balance 'interesting' challenges with your 'day to day' work?
4. Are you challenged by other motivated programmers around you?
5. Do you have a broad field of interest that informs your work?

ACCU 2011 Conference

Chris Oldwood shares his experiences.

This year's ACCU conference (my fourth) was always going to be a little bit different because *this year* I was going to be speaking. That meant no stupid late nights staying up in the bar chatting with my fellow ACCU members and other like minded individuals about programming because I was a professional and I had a job to do…

I arrived on the Tuesday evening just in time for the obligatory visit to Chutneys – an Indian restaurant in the centre of town. Many of the usual suspects were already there and it took no time at all to slip into idle chat about what we've all been up to since we sat in the same restaurant a year ago. Then it was back to the bar and a chance to catch up with the late arrivals and gatecrash a few conversations in an attempt to do a little networking before a nice early night. What a ridiculous notion. Of course I ended up in the bar until 4 am in a discussion about equality and value identity – John Lakos was there! Still, I wasn't on until Friday afternoon so there was plenty of time to recover.

The opening keynote on Wednesday morning was from Giles Colborne, titled 'Advanced Simplicity'. As someone who is a backroom boy these days I don't spend any time doing UI work, but the talk seemed to transcend that and provided perfect analogies that could just as easily be applied to API design. He provided some nice examples of how you can avoid a whole class of potential error scenarios by just using a simpler control in the first place. There were also examples of ideas that you would expect to work, but didn't, and cases where people used the tool in ways the designers hadn't anticipated. Some quotes from the likes of Donald Norman and Alan Cooper provided a nice backdrop.

That was the easy bit out of the way and a shot of coffee was then needed whilst I spent the 30 minute break trying to decide between 5 typically awesome sessions. Sometimes you just have to use darts, dice, a coin or whatever because the line-up is so varied. For my first session I chose Peter Pilgrim's 'Introduction to Scala'. I don't generally follow the Java world but I have heard interesting things about Scala and Peter's introduction was a whirlwind tour of the various language features. It has elements of functional languages that I'm not used to but his examples still made sense and he finished off with a nice comparison of the various JVM based languages that seem to be vying for the hearts and minds of developers.

The 90 minute lunch break is quickly over when you've been nattering away and it was Lisa Crispin and her 'Dealing with Defects' talk that filled my early afternoon slot. This was an interesting look at the techniques and tools that teams often use to track and manage bugs. There was definitely a feeling that many organisations spend too much time on tracking and managing bugs and not enough on preventing them in the first place. Lisa got the audience to do a little exercise where we played the role of developer and tester. The two had to collaborate to draw a simple map, first standing back-to-back and then facing each other. This illustrated how much better it is for the two to engage directly instead of via, say, email – simple but very effective. This was a useful talk that helped to confirm my team are heading in the right direction.

I made Kevlin Henney's talk about OO as my final choice for the day because I saw his 'Value Objects' session last year and I clearly still have much to learn about OO even after all this time. He didn't disappoint this time either as he slowly managed to show that the SOLID principles rest on flaky ground, ending up with just SID! However I got the feeling that the Single Responsibility Principle (the S in SOLID) should also be taken under advisement. His revelation about the Liskov Substitution Principle was most enlightening and goes to show that the terms type, subtype, class, interface, abstract data type, etc. are often banded about and incorrectly used interchangeably in OO circles. The question now is whether I still answer 'Encapsulation, Polymorphism and Inheritance' in a job interview?

Sadly Uncle Bob couldn't provide the closing day's keynote and so Michael Feathers stepped up to the plate instead. This was a curious keynote that I felt never really went anywhere. He commented on the productivity of Ruby and noted the verbosity of C++ but never appeared to come to any conclusion. Was he suggesting the world would be a better place if we all switched to Ruby? Did the fact that there were so many C & C++ programmers in the audience cause him to refrain from saying what he really wanted to? If nothing else it generated chatter in the bar after…

Tom Preston-Werner, one of the big cheeses at Github, opened business on the Thursday morning. On the one hand it's nice to see that some companies can provide an excellent place to work, but on the other there is a large degree of smugness that goes with this kind of presentation. Unmetered holidays, self organising teams, a bar on-site, and they're making a profit too – it sounds like programming nirvana. It sure will be interesting to see how it scales with the age of the employees and the size of the company.

My first optional session for the day was titled 'Distributed Computing 2.0' by Filip Van Laenen. I wasn't sure what to expect from this as I didn't really read the blurb and therefore was somewhat surprised to hear the details of his hobby project that used Twitter as the communication channel for a distributed system. This immediately raised many questions in my mind about scalability and reliability and he answered nearly all of them during his talk as he explained how Facebook and Yahoo/Google Groups could also be used in a channel bonding style. The problem he is attempting to solve fits nicely inside the limits of the architecture; ok, so it's not a general purpose solution, but he never claimed it was. I soon started thinking about how to use the add-on services like yfrog and twitpic to overcome the message size limits so it's clearly had some impact.

I suppose I didn't really have a choice about what I was going to see in the afternoon. Given that I had stayed up in the bar into the wee hours Tuesday night to listen to an argument about equality I just had to go and see Roger Orr and Steve Love cover the subject in detail. This focused mainly on the pitfalls of how to correctly do an equivalence comparison in Java and C# as there is much complexity involved. C# has a plethora of interfaces for dealing with equality – more than I knew. They also showed some really 'interesting' behaviour around the way one JVM caches the boxed values –100 to +100. It's not often that people remark how simple C++ is, but in this case it really is and the wording in the standard backed that up.

I felt I needed to hear some hard-core techie stuff at some point and so Scott Meyers was the perfect choice with a session all about CPUs and cache lines. Scott is an entertaining presenter and he made the low-level topic easily understandable. Although I was aware of many of the symptoms and solutions from Herb Sutter's writings I gained a much better insight into the hardware side and why these problems exist. I have much more respect for the chip designers now! He also briefly covered Profile Guided Optimisation and Whole Program Optimisation to cover the instruction cache side of things.

Lightning talks – 5 minute talks by the general populace – made their appearance again this year and their popularity is clearly growing as the

CHRIS OLDWOOD

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

Code Critique Competition 69

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

What's the best way to read output from Fortran fixed-format strings using **scanf**? The example is a database file with lines such as

```
  "  26  2996100  1"
```

which were written by a fixed width format (i.e. this ***should*** be interpreted as 'leading space, _26, __2, 996, 100, __1').

The problem is that automatic whitespace skipping in **scanf** means that any **%d** format string is almost guaranteed to get confused for one or other variant of full fields, e.g. the 'obvious' **" %3d%3d%3d%3d%3d"** format string reads **26 299 610 0 1** here.

Listing 1 is a simple program demonstrating the problem (and the asymmetry of C input/output formats!), and the results are:

```
C:\cc68>output | input
1 2 3 4 5
100 200 300 400 500
26 299 610 0 1
```

(Thanks to Robin Williams for suggesting this issue's critique.)

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

ACCU 2011 Conference (continued)

number of speakers approached 23 over the two days. Olve Maudal kicked off the first set of talks with a tongue-in-cheek piece called 'Technical debt is good!' Other notable shorts include Pete Goodliffe's 'Manyfestos', which was a poke at the ever growing number of developer manifestos out there, and Mark Dalgano's 'Optimise for Unhappiness' which reflected back on the day's keynote to show how too many companies treat their employees.

Friday's keynote was due to be given by both authors of the hugely popular book *Growing Object Orientated Software*, but unfortunately it was left to Steve Freeman to go it alone. The title, 'Good Enough Is the Enemy of Better', sums up perfectly how software development is often approached in many organisations. But Steve showed how it can be a false economy and that doing it right can pay dividends both from an operational and maintenance perspective. As someone who is currently working on a system built this way it's great to be nodding in absolute agreement instead of longing to be taking part.

The morning coffee break was swiftly followed by a visit to see Jon Skeet discuss Java and C# – the lessons learned and mistakes replicated by the latter from the former. Jon may be a Java programmer by day and C# hobbyist by night, but he still knows way more about C# than most! The discussion about the differences, such as the handling of generics was typically enlightening as was the curious adoption of much of the Object behaviour and how that was probably a bad choice. I left somewhat disappointed as I had hoped for more crystal ball gazing and the answers to some of his questions where clearly already in 'C++' and 'D' which just goes to show how hard it is to keep up with even the 'curly brace' languages.

The weather really brightened up for lunch and it was somewhat of a shame to be stuck indoors in the afternoon, but if anyone is ever a ray of sunshine it's Tom Gilb. I thought I should go to at least one 'process' related talk and the contentious title 'What's Wrong With Requirements' sounded like it could generate some debate. Tom's premise always seems to be that everything can be measured and quantified and this was no different. What seems to be wrong is that we use 'woolly' terms instead of backing them up with a measurable value to guide the design and implementation. A simple message but fairly well rammed home with plenty of entertaining anecdotes and tangents that only Tom seems to get away with.

I really had no option about what to go to after the break because I was one of those going to be doing the talking! It was a great experience that I felt went pretty well, even if I did speak a little too quickly. Interestingly however feedback suggests that I need to do work in other areas instead; such is the problem with not being able to watch yourself. The good thing is that I have 12 months to work on my presentation skills before next year.

The second round of lightening talks followed the day's final coffee break and once again it was a packed with speakers. Didier Verna is clearly in the wrong job and should be 'treading the boards' as he appeared again – twice. Richard Harris rose to the challenge from Roger Orr and Steve Love by showing us how to correctly compare floating point numbers and Jim Hague tried to drum up interest for the local folk dancing display before showing what he'd unearthed whilst spelunking the King James' Bible. Phil Nash closed the year's talks with a very succinct demonstration of his new lightweight unit testing framework called Catch.

Friday always plays host to the Speakers' Dinner where the attendees get to spend a little more time up close and personal with the presenters. After each course they have to switch tables whilst the speakers stay put so there is always someone new to chat with. There was also an auction in aid of Bletchley Park that raised a few thousand pounds and the announcement of the handing over of the conference chair duties from Giovanni Asproni to Jon Jagger – a tough act to follow! The formal proceedings finished around 11 pm and so the bar played host to those that wished to continue the festivities.

Sadly my presence ended a little earlier this year and so I had to miss out on the Saturday sessions. I haven't even looked at what was on because it will only depress me further knowing what has escaped me. I know I say it each year, but you really cannot put a price on the time you spend with the speakers and other attendees in the bar at the end of each day. Yes, as the evening progresses the value-per-pint slowly diminishes, but even at 4 am after going round the houses a few times it's good to know that even those you would consider at the top of their game still have much to debate.

```
//-- output.c --
#include <stdio.h>
int process(int v1, int v2, int v3,
  int v4, int v5)
{
  printf(" %3i%3i%3i%3i%3i\n",
    v1, v2, v3, v4, v5);
}
int main()
{
  process(1, 2, 3, 4, 5);
  process(100, 200, 300, 400, 500);
  process(26, 2, 996, 100, 1);
  return 0;
}
//-- input.c --
#include <stdio.h>
int main()
{
  while (!feof(stdin))
  {
    int i1, i2, i3, i4, i5;
    scanf(" %3i%3i%3i%3i%3i\n",
      &i1, &i2, &i3, &i4, &i5);
    printf("%i %i %i %i %i\n",
      i1, i2, i3, i4, i5);
  }
}
```

Critiques

Pete Disdale < pete@papadelta.co.uk>

Q. ‘What’s the best way to read output from Fortran fixed-format strings using `scanf()`?’

A. ‘Don’t. Next question?’

OK, that’s rather terse but does convey, I believe, generally sound advice. I would even go as far as to suggest that using `scanf()` for any input where that input cannot be 100% guaranteed is a Bad Idea as it can leave trailing `\n`s and other garbage at end-of-line and confuse subsequent `scanf()` calls on that input. Much better to use `fgets()` [which will read the whole line of input and eat the newline if there is one and the input buffer is large enough] and then use `sscanf()` on that buffer. Now that the rant is done, I suppose I should provide an alternative...

The input as given is, with the exception of the leading space character, typical of SDF files that can be produced by dBASE and early spreadsheet programs like Supercalc – so not limited to Fortran output. Over the years I have found the most reliable approach to reading these output files is:

- read an entire line of input using `fgets()`
- knowing the layout, extract each field one at a time from the input buffer.

Given the example file, with 5 integer fields of width 3 digits each, one approach might be something like this:

```
#include <stdio.h>
#define FWIDTH  3 /* field width */
#define NFIELDS 5 /* nr of fields */
int main()
{
  char line [BUFSIZ];
  int  v[NFIELDS];
  while (fgets (line, BUFSIZ, stdin) != NULL)
  {
    int len = strlen(line);
    int has_newline =
      line[len - 1] == '\n' ? 1 : 0;
    char *p =
      &line[len - has_newline - FWIDTH];
    int i;
    for (i = NFIELDS - 1; i >= 0; i--)
    {
      v[i] = atoi(p);/* safe for 3 chars */
      *p = '\0', p -= FWIDTH;
    }
    printf("%i %i %i %i %i\n",
      v[0], v[1], v[2], v[3], v[4]);
  }
  return 0;
}
```

Note that this processes the buffer backwards (right-to-left); it could be processed left-to-right starting at (line + 1) but this would require saving and replacing each char position that has to be replaced with a `\0` for the `atoi()` to work. It also handles the case where there is no `\n` at the end of the input (typically the last line of the input stream) though otherwise error checking is minimal...

But to get back to the original question, it ***is*** possible to use `scanf()` (or rather `sscanf()`) to process the input, using `%3c` to parse the input into strings and use `atoi()`/`strtoi()` to convert to integer:

```
#include <stdio.h>
int main()
{
  char line [BUFSIZ];
  char s1[4], s2[4], s3[4], s4[4], s5[4];
  /* sscanf(...%Nc...) does not null-terminate
     the string */
  s1[3] = s2[3] = s3[3] = s4[3] = s5[3] =
    '\0';
  while (fgets (line, BUFSIZ, stdin) != NULL)
  {
    /* skip past leading space */
    sscanf(line + 1, "%3c%3c%3c%3c%3c\n",
        s1, s2, s3, s4, s5);
    printf("%i %i %i %i %i\n",
      atoi(s1), atoi(s2), atoi(s3),
      atoi(s4), atoi(s5));
  }
  return 0;
}
```

This appears to work for the implementation of `sscanf()` that I used, but it would be a brave person that assumed that it would work on all implementations.

Summary: ‘Don’t use `scanf()`.’ Read a whole line of input and use `char` pointers with `atoi()/strtoi()` to do the conversion to integer. If for no other reason, such an approach affords the programmer the scope for input validation and error checking (for example that the parsed strings are actually composed of digits)

Commentary

This critique is straight C and highlights one of the major shortcomings of `scanf`: white-space handling. (Note that trying to solve the problem using by C++ doesn’t help because although the C++ iostream library provides some additional control over white-space handling it does not make handling this example any easier.)

In general the approach of reading data using `fgets()` and then splitting up the lines of input as strings is usually a good solution; although if this is a problem you face often there are some libraries that provide richer input processing than standard C.

People are often worried about the performance overhead of processing each character multiple times. This is not usually a problem in practice as the cost of reading the characters, typically from disk or across a network, into the process’s memory usually dwarfs the differences between the different types of input format processing.

The winner of CC 68

The winner of this code critique is of course Pete Disdale. He provided two different ways to approach the problem and I thought explained the underlying issue clearly. I also liked his tongue-in-cheek first answer to the question: 'Don't. Next question?'

Code Critique 69

(Submissions to scc@accu.org by Jun1st)

I've written a streaming helper for dates – seems to work OK – and a manipulator so I can stream today's date, plus a convert from string function. Please can you review my code?

Here is an example of usage:

```
#include "strdate.h"
void example(time_t tv)
{
  std::cout << date(tv) << std::endl;
  std::cout << strdate << std::endl;
  tv = date::convert("2000-01-01");
}
```

The code is in Listing 2 and a small sample program in 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.

Listing 2

```
#include <iostream>
#include <time.h>
#pragma once
// date class
class date
{
  time_t const& tv;
public:
  date(time_t const& tv) : tv(tv) {}
  // print self to stream as YYYY-MM-DD
  void printOn(std::ostream& os) const;
  // convert YYYY-MM-DD to time_t
  static time_t convert(char const *);
};
void date::printOn(std::ostream& os) const
{
  int day = tv / 86400;
  // base on Mar 1968 (makes leap years easy)
  day += 365 + 366 - 60;
  int year = day / 365;
  day -= year * 365;
  day -= year/4;
  if (day < 0)
  {
    day += 365;
    year--;
  }
  int month = 0;
  int daycounts[]
    = {31,30,31,30,31,31,30,31,30,31,31,28};
  while (day >= daycounts[month])
  {
    day -= daycounts[month++];
  }
  // base on Jan
  month += 2;
  year += month/12;
  month %= 12;
  os << year+1968 << "-";
  os << (month<9?"0":"") << month+1 << "-";
  os << (day<9?"0":"") << day+1;
}
```

Listing 2 (cont'd)

```
time_t date::convert(char const *date)
{
  int yr,mn,dy;
  if (sscanf(date,"%i-%i-%i",
      &yr, &mn, &dy) < 3)
    return -1;
  // base on Mar 1968
  yr -= 1968;
  mn -= 3;
  if (mn<0)
  {
    yr--;
    mn += 12;
  }
  dy--; // 1-based
  int daycounts[]
    = {31,30,31,30,31,31,30,31,30,31,31,28};
  while (mn)
    dy += daycounts[mn--];
  dy += yr * 365;
  dy += yr / 4;
  // rebase to 1970
  dy -= 365 + 366 - 60;
  return 86400 * dy;
}
// stream date
std::ostream& operator<<(std::ostream& os,
  date const& rhs)
{
  rhs.printOn(os);
  return os;
}
// stream today's date
std::ostream& strdate(std::ostream& os)
{
  return os << date(time(0));
}
```

Listing 3

```
#include "strdate.h"
int main()
{
  std::cout << date(86400) << std::endl;
  std::cout << date(86400*31) << std::endl;
  std::cout << strdate << std::endl;
  std::cout
    << date(date::convert("2000-01-01"))
    << std::endl;
}
```

Desert Island Books

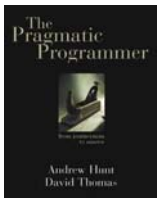
Chris O'Dell makes her selection.

I first became aware that Chris O'Dell had joined the ACCU when she submitted a piece for the CVu I was editing. Since then Allan Kelly and I have been nagging her to do various things for the ACCU and this is the latest. I'm also glad to say that Chris has become a regular at ACCU London events.

It feels like the ACCU has not had much in the way of new blood for a little while and it really needs the contributions from people like Chris. I would like to try and address this, so if you have joined the ACCU in the last few months or years, please drop me a line (my email address is at the bottom of this column).

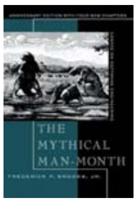
Chris O'Dell

When Paul suggested I write my selection of Desert Island books for CVu I was initially flattered, then immediately panicked. This was due to two reasons; the first is that I've only been coding professionally for just under six years and as such I'm still reading my way through a lot of the standard books most programmers should read and I felt that my selection would not be particularly interesting or new. The other reason is that I have a nasty habit of not finishing the books I start. I will pick up a book, read the first few chapters and then become distracted by the next 'must read' book and immediately start that one without finishing the first. In fact, I have a large pile of books that I am yet to finish or even start. I have even taken to repeating to myself 'Don't buy any more books until you've finished the ones you've got' over and over like some sort of mantra.



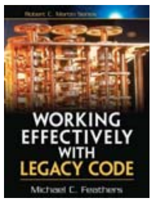
One of the books that has had the greatest impact on my day-to-day coding is *Pragmatic Programmer*. I remember reading each section and thinking 'that's so much clearer now' or 'I knew I was on the right track'. As such, I would happily take this with me to my desert island because I feel re-reading it would be a useful way to bring myself back down to earth and look for a pragmatic approach to my situation. Immediately after finishing *Pragmatic Programmer* I read *Mythical Man Month*, which also put a lot of pieces into place for me.

A little secret of mine is that my first job out of University was as a 'Junior Project Co-ordinator'. I was offered it before I had even officially graduated and so I jumped at the chance to be earning money and starting to pay off my debts. Unfortunately for me, I didn't take the time to question why I was offered a non-programming role despite interviewing for one and turning up with the full source code of my C++ and OpenGL 3D game – I was offered a job and so I took it. Of course, I found myself dissatisfied. Project Management was not what I wanted to do, for me it was not fulfilling and I didn't care that someone else must have believed that I did not belong in the (strangely all male) programming team. After a time I demanded that I be moved to the Programming team and eventually, through my own persistence, I was moved teams and began learning C#.



Through this experience I can now say, categorically, that I do not ever want to walk down the Project Management path, although I did learn a lot from that side of the business, e.g. How customers can, through no fault of their own, miscommunicate and misunderstand their own requirements, why the business needs visibility of the Development team's progress or estimates, and more importantly, how to communicate with all levels of the business. As such, I felt that *Mythical Man Month* understood many of the problems I had experienced on a project and that had been experienced before, time and time again, even researched, analysed and written up for everyone to read. Many of the findings have stuck with me, such as the exponential slowdown that occurs when you add more and more developers to a project already underway. I'll take it with me to ponder over again and hopefully I will be stranded with a project manager and I can lend it to them in the hope that they may in future adjust their Gantt charts more appropriately.

I remember reading *Code Complete 2* very early on in my career, but these days I cannot recall anything specific from it. Maybe I should read it again now that I've got a few more years under my belt, but I think I would rather take a book which is currently in my bag at all times – *Clean Code*. So far I am finding the advice within it to be extremely practical, simple to follow and yet with the biggest impact. The promotion of clean code inspires a sense of pride in your work as though it is your honour to leap into existing code bases and make them more beautiful places when you're done. I feel that I will enjoy finishing this book whilst in the shade of a palm tree.



Another book that I have started but not yet completed (mostly because I picked up the shiny new *Clean Code*) is *Working with Legacy Code*. I'm about a quarter of the way through and enjoying what I've read so far. I can see how test driven development evolved out of this book and others like it. The idea of gradually wrangling legacy code into something testable has given me greater confidence and tools for dealing with such systems in my every day work. It is always a good idea to remember that once code has been written it almost instantly becomes a 'legacy system' – the addition of testing around it gives future developers (including your future self) the confidence to work with the code without introducing unknown bugs and breaking existing behaviour.

Whilst I'm on this desert island I might as well take some books from the pile of those I have not yet started to read. I'll start with the Gang of Four's *Design Patterns*. I have been subconsciously putting off reading this because I have been told that it's very dry and difficult to get through, but if it's just me, the sun, sand and the book then I'm sure I'll make my way through it and that I'll be better off for it. I'll also pack *Refactoring* from the pile as I'm hoping it will tie in nicely with *Working with Legacy Code* and leave me feeling ready to take on all legacy systems when I'm finally rescued.

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.

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)

Beginning F#

By Robert Pickering, published by Apress, ISBN 978-1430223894

Reviewed by Joes Staal

Not recommended

I found this book boring and not very well structured. The author is clearly enthusiastic about F# – he continually says how great and wonderful the language is, but doesn't convince me. He introduces a lot of ideas without explaining why and it was not before long I felt overwhelmed by the amount of material he tried to cover. It was very difficult to see the wood for the trees.

The book starts with explaining how to use F# for functional programming, imperative programming, object-oriented programming and usage of the F# libraries. It then delves into more specific topics such as user interfaces, data access (XML, ADO.NET, LINQ), parallel programming and language oriented programming. For me it didn't work out. It was just to hard to get over the introductory chapters and into the more specific ones. I've started the book three times over and was in the end defeated. I didn't get further than finishing chapter 7, out of a possible 14.

Programming F#

By Chris Smith, published by O'Reilly, ISBN 978-0596153649

Reviewed by Joes Staal

Recommended

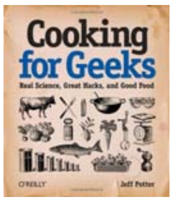
Because I really wanted to learn F# I decided to try this book by Chris Smith. Although first 6 chapters titles are the same as in Pickering, this book is completely different.

Bookshops

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

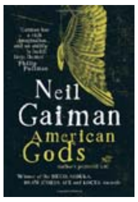
- **Holborn Books Ltd** (020 7831 0022)
  www.holbornbooks.co.uk
- **Blackwell's Bookshop**, Oxford (01865 792792)
  blackwells.extra@blackwell.co.uk

Desert Island Books (continued)



A book which I'm going to count as technical, albeit tenuously, is *Cooking for Geeks*. A Christmas gift from my boyfriend as he knows full well my inability to cook yet refusal to give up experimenting. It contains interesting and useful facts about cooking from a more scientific point of view such as what temperature and time foods should be cooked to ensure all 'bugs' have been removed, which I believe will come in extremely handy when I try to make a meal out of the island's wildlife.

Now that we've got the tech books picked out the tough part is picking just one novel. I love to read sci-fi and fantasy and as anyone who reads these genres knows the stories tend to span quite a few physical books. My favourite series is the *Deverry Cycle* by Katharine Kerr but it encompasses sixteen books and I think that might be stretching the rules just a little bit. I also love Neil Gaiman's *Sandman* graphic novels but that's still been compiled into ten volumes. Isaac Asimov's *Foundation Series* is seven novels and Joe Abercrombie's *First Law* is a trio of books. So, I stare at my bookshelves mumbling to myself 'just one book...'. I first pull out Terry Pratchett and Neil Gaiman's *Good Omens*, which I bought for 50p from a local library clearout when I was in school. Then I pull out *Neuromancer* and recollect my joy at the discovery of cyberpunk, which also reminds me of Snowcrash and Philip K. Dick's *Do Android's Dream of Electric Sheep*. Finally I settle on Neil Gaiman's *American Gods* remembering how the story made me laugh and cry and that rereading it will hopefully keep me amused during the lonely days on my island.



As for the music I must first wonder what I am to do with my CDs on the island? I am guessing that I won't have the foresight to bring a CD player with me or that the island comes complete with power sockets to keep it running. Maybe it would be more appropriate to bring my iPod packed full of all my music and some sort of solar powered charging device, but again, that would be stretching the rules and ruining all the fun.

One of the first albums I ever bought was *The Bends* by Radiohead, the angst ridden songs carried me through high school and I believe that it will help me work through my feelings during moments of helplessness on my island.

Another album that I can listen to over and over again is the Eels' *Daisies of the Galaxy*. I realise that in the November issue Alan Stokes had already mentioned this, but it was the soundtrack to my university years and even now some of the lyrics can still evoke an emotional response.

As always, I am open to receiving more book recommendations, although, I shan't be buying them until I've finished the ones I've got. I shan't buy any more books until I've finished the ones I've got. I shan't...

Next issue: James Byatt

View From The Chair

Hubert Matthews
chair@accu.org

There has been an interesting discussion on accu-general recently about what people would like to change about the ACCU. This is very useful input for the committee and, fortunately, aligns well with our own thoughts. The primary focus seems to be marketing the ACCU better and revamping the web site. Once the AGM is out of the way, the newly elected Treasurer and Secretary are safely ensconced and, I hope, the financial situation improved through the fees increase we can focus less on immediate issues and more on the longer term.

The web site is our shop window to the world and we would like to improve it significantly. We have created a prototype site based on Wordpress that allows members to post their own content, articles, book reviews, etc as well as aggregating the blogs of various members. One useful feature allows members to post events to a calendar and we hope that this will be used to publicise not only ACCU events (such a local group meetings) but also other non-commercial events of interest to members such as those run by organisations like the BCS or even commercial companies. The existing book reviews would be migrated to the site as ordinary articles, thus allowing both the on-site search and external search engines such as Google to find them. The ACCU used to be known for its high-quality book reviews and we should look to live up to and enhance that reputation. The key focus will be to provide regular, fresh, rich and interesting content and to achieve this by making it easy for members to contribute. We have a wonderfully diverse and interesting set of people, so let's make it easy for them to contribute and showcase our collective talents. Using a base such as Wordpress means that we can achieve a lot at a low cost and also benefit from the large number of plug-ins and themes that are available. A more relevant site will draw more traffic and therefore be more attractive to advertisers.

What will it take to achieve this vision and how can members help? We are looking for volunteers to help in the following areas:

- look-and-feel (Wordpress themes and CSS)
- gentle Wordpress set up and hacking
- content creation and editing
- ideas for what you want to see on the site
- testimonials from members as to what value the ACCU brings to them
- data and content migration from the existing site
- functional and security testing
- integration with advertising and payment systems

Contributions can be both small or large and at whatever level or to whatever extent people feel they can help. If you want a site that meets your needs then come and join us. The overall effort will be coordinated by the committee but we do not have the broad range of skills required to do everything. If you have ideas about what you'd like to see or can offer some assistance please drop a line to one of the committee - it's your site so help us make it something we will use, a site we can all be proud of and something that will attract new members.

Bookcase (continued)

For me, Chris Smith introduces concepts at the right pace and gives the information I need at that moment. More details are given later when needed. Chapters 7 and 8 cover applied FP and applied OOP. After these chapters the reader is able to think and program in F# and the second part of the book introduces scripting, computation expressions (also called workflows), parallel programming, reflection and quotations (a way to perform code analysis and inspection, allowing computation to other platforms).

When studying both this book and Pickering, I decided to work on some problems from Project Euler and see how much I understood of the material I digested. With Pickering's book I had trouble moving from imperative programming towards a functional style, but with Smith's book I was able to write some 'acceptable' F# code. Still, I am a beginner and need to hone my skills. And although I liked the book by Chris Smith, I am still looking for the definitive F# guide – which should include exercises, something both books are lack.

Dive Into Python

By Mark Pilgrim, published by Apress, ISBN 978-1590593561

Reviewed by Paul Grenyer

I used to do the odd bit of Python programming a few years ago and as part of a recent position. I hadn't done any for a year or more and when I came to pick it up again, to write some acceptance tests for a .Net command line application, I found I could hardly remember anything. So I asked some people about good refresher books. *Dive Into Python* came dubiously recommended.

Dive Into Python is actually a great book, once you get used to the chatty style and things like '... actually, what I just told you was a lie. The truth is ...' and '... you didn't really think that did you? Go and sit in the naughty corner!' Each chapter builds on the next and has copious examples, all of which are explained in detail line by line. The domain of all the examples are simple and extremely well thought out.

The web service chapters aren't really about web services, but web service clients. I found this rather disappointing. I skipped the chapters on XML manipulation and regular expressions. The chapter on dynamic functions says that you should read the chapter on regular expressions first, but there's really no need as all the regular expressions are explained in detail. The unit testing chapters are very good. The functional chapter is really just an extension of the unit testing chapters. Following the optimisation chapter, the final fifth of the book is appendices.

Dive into Python is available in its entirety at http://diveintopython.org/ or in hardcopy published by Apress.

Introduction to the Boost C++ Libraries; Volume I - Foundations

By Robert Demming and Daniel J Duffy, published by Datasim Education BV, ISBN 978-9491028014

Reviewed by Paul Floyd

Recommended

Though Boost is a large federation of libraries, this book is mercifully concise and to the point. It's the first book that I've read from this publisher. The page layout is quite basic – no fancy fonts, icons or sidebars.

In the name of brevity, the authors attack Boost straight off. There could have been more on getting and installing Boost, which has a fairly idiosyncratic build system. If you have an OS that installs Boost by default (and you are satisfied with that version and not hungry for the very latest version of Boost), then this is not a problem.

I haven't yet tried out any of the examples, but they look well balanced. Not all of Boost is covered (hence the 'Volume I'). The parts that were chosen seem to fit together quite well and generally cover data/smart pointers, serialization, threads and mathematics.

At present I don't use Boost in my day job. I wanted to read this book to maintain my knowledge of Boost. In the end, I felt that my goal was achieved.