The magazine of the ACCU

www.accu.org

{cvu}

Volume 20 Issue 6 January 2009 £3

Features

{cvu}

Volume 20 Issue 6
January 2009
ISSN 1354-3164
www.accu.org

Good Intentions

At this time of year it seems I should be following the high street trend of enclosing a free gift to encourage loyal readership. But I know you better than that. *Newsweek* and *Digital Photography* are brushed aside when C Vu pops through the door – and with good reason! We've got a great selection of material in this issue, from Paul's excellent boiler plating article to Colin's extensive review of Fortran literature spanning 27 years. We've also got a fascinating insight into the difficulties with dates, and a great guide to exception handling for those of you that have so far managed to avoid this all-too-often 'optional' area of C++.

For everyone eagerly awaiting the 2nd instalment of the PC Lint guide (myself included), I'm afraid that it has been slightly delayed, but it will be featured in the March issue, all set to be hosted by Roger Orr.

What I love about this time of year is the feeling of potential that the New Year brings. Whether it's official resolutions or just a mental reminder to do something differently, there is something very exciting about planning how you're going to fill a brand new set of 52 weeks (more running, less Xbox).

If you're in need of ideas, the ACCU conference is now firmly on the horizon (details available at www.accu.org/conferences), so if you've never been before, now is a good time to start thinking about booking your place (or convincing your boss to book your place).

I'd like to say a huge thank you to everyone who worked so hard in the always-hectic run up to Christmas, to ensure that you, the reader, now has something to fill those long January nights with. Fortunately, I've already read this issue, so I'll leave the rest to you – enjoy, and have an action-packed 2009.

FAYE WILLIAMS
GUEST EDITOR

The official magazine of ACCU

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

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

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

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

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

DIALOGUE

FEATURES

SUBMISSION DATES

C Vu 21.1: 1st February 2009
C Vu 21.2: 1st April 2009

Overload 89: 1st January 2009
Overload 90: 1st March 2009

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

Santa Claus and Other Methodologies

Gail Ollis takes a festive approach to software development.

I don't believe in methodologies. This rather contentious position has opened up many interesting conversations about approaches to software development, but without the subsequent conversation it is a very bald statement. So what do I really mean by it, and does examining it closer actually uncover something more constructive? In this article I will outline my position in more detail and explain how, in the course of bitter experience, I came to reach it. But perhaps 'bitter' is the wrong word, because I have had useful opportunities to observe the relationship between plausible methodologies and the real world. So while I will explain some of the reasons for their failure to solve problems, I will also look at the essential elements of successful solutions and how – somewhat to my surprise – I found that methodologies have a valuable contribution to make, much like Santa Claus contributes to the magic of Christmas despite the impossibility of flying reindeer.

Terminology (another -ology!)

Perhaps I should begin by defining my terms. First the sometimes ticklish question of 'methodology'. The suffix '-ology', defined by the Oxford English Dictionary (OED) as 'the science or discipline of', is widely understood to mean simply 'study of'. Thus in its software context 'methodology' is often regarded as a pretentious and inaccurate inflation of 'method'. Those who remember BT's 'exam results' TV advert of the 1980s will know that '-ology' carries an aura of academic respectability. Maureen Lipman plays a grandmother who phones to congratulate her grandson on his exam results, only for him to report that he has failed – no passes except for pottery and sociology. 'He gets an Ology and says he's failed!' she exclaims. 'You get an Ology, you're a Scientist!'

Regardless of how 'methodology' came to be used for a systematic collection of activities, this meaning is now enshrined in the OED:

> Originally: the branch of knowledge that deals with method generally or with the methods of a particular discipline or field of study; (arch.) a treatise or dissertation on method; (Bot.) systematic classification (obs. rare). Subsequently also: the study of the direction and implications of empirical research, or of the suitability of the techniques employed in it; (more generally) a method or body of methods used in a particular field of study or activity.

Those who approve of the term will not be surprised to find its software meaning catered for in this definition, while its detractors can perhaps take comfort from its position right at the end of the list, after obsolete and archaic usages. Unfortunately, this usage also means that the study I am undertaking here must surely be in the field of methodologyology!

As for not believing, I do not mean this in the same way as 'I don't believe in Santa' – I do, of course, believe that methodologies exist! Nor do I mean that I think they are worthless or never work. I am, however, a complete unbeliever in the almost religious zeal that sometimes promotes them as the industry's salvation. In short, I don't believe the hype. I don't believe in applying a solution without first analysing the problem; at best it is unhelpful and at worst counterproductive. The reasons for failure to deliver on the promises may lie in the soundness of the method, its 'fit' to a particular problem or culture, the means of its implementation – or any combination of these. I will now examine two examples which demonstrate these failings; examples which, no longer being flavour of the month, will hopefully not risk much offence. One is a methodology specifically for software development. The other is for project management and was employed on a large, interdisciplinary system development. Both contribute material to the analysis that follows them of how methodologies can fail to deliver.

Example 1: RTSASD

My software example is Real Time Structured Analysis and Structured Design (RTSASD), which I used in a two-site development of a large new radar system in the late 1980s. This system was a radical advance from the earlier systems in which software played a largely passive role of processing the data from the antenna, requiring as it did additional software to perform the complex task of guiding a 'steerable' radar beam. Most of the code was written in Ada for Intel 486 processors, but low level signal processing required the greater power of a Distributed Array Processor (AMT DAP 510). RTSASD is not language specific, or at least not for the languages of its day, which predated the commercial rise of object oriented software. It is a method which maps problem domain functions to functional modules in the design; decisions are made about the scope and interfaces of components in the analysis stage and strongly influence the design. Looking back on this, two alarm bells ring immediately.

Firstly, I have grave doubts about the quality of analysis that can be achieved by such an implementation-focused approach. It is a flawed perspective for undertaking analysis of a customer's requirements; whenever I have seen it adopted (and its adherents are widespread) it has produced a system which reflects the developer's view of how to build software and not the customer's view of his business. Coupled with an almost litigious devotion to contractually casting the requirements in stone as the ink is drying on the contract, the net result is, time and again, a system which meets the letter of the requirements, ticks all the boxes of the contractual acceptance tests but leaves the customer less than delighted with a purchase that fails to achieve what he had envisioned.

Secondly, such early design decisions flout the principle of reversibility: 'There are no final decisions' ([1], pp.44-46). In RTSASD, the analysis phase makes premature design decisions which constrain the implementation. In fact, an analysis formed in terms of the implementation not only denies flexibility for the detailed design, but also limits the potential expressiveness of the analysis. The solution is put before the problem.

Further issues become evident only upon considering the artefacts produced to represent the system pictorially (data flow and state transition diagrams) and textually (data dictionaries and process specifications). Without doubt these captured relevant information and proved a better vehicle than wordy documents could have done in facilitating collaboration between the two sites. Where they failed was in mapping to well-structured code; it is perhaps not surprising, then, that some of what was written managed quite spectacularly to break the compiler. Nor did this style of representing the system support partitioning for

GAIL OLLIS

Gail was a software developer for 20 years and cares how the job is done. She wondered 'why did they do it that way?' once too often, so now she's studying psychology. Send offers of paid research studentships to accu@ollis.org.uk!

work on subsystems, leading to whole groups of data prefixed with the name of the site, subsystem and purpose (e.g. `xxx_yyyyy_zzzzzzz_`) to package them together and create a namespace which the artefacts, in their very flat view of the data, could not manage for us.

Despite the emphasis on functional aspects, the pictures of the system did not map readily onto Ada packages and files, so this was awkward to do and the rather haphazard transition was one-way – it was not always clear where elements of the code had derived from, so the RTSASD artefacts became a disposable step as part of the process rather than a body of documentation that helped make sense of the code. The step from CASE (Computer Aided Software Engineering) tool to code was reminiscent of the simple box in the midst of a complex mathematical formula in an old cartoon (by Gary Larson, I believe): 'magic happens here'.

Creating the artefacts also became an end in itself. Following this approach required a tool, and that tool was Teamwork. Persuading it to accurately convey our design took so much effort that you could be forgiven for thinking it was so named because it caused the team so much work. It was a classic case of the tail wagging the dog as the creation of artefacts became dominant over the important work of *thinking*. Somewhat scarred by the experience, I would be wary of any tool with 'team' in its title; teams are composed of human beings and interposing a computer is not necessarily the best way to get them working together. (That's not to say that computer-mediated communication can't be beneficial – but that's a discussion for another article).

some people seem to be satisfied with Lego, smoke and mirrors

What *were* we thinking? Given the complexity of the task it seems a particularly risky time to have attempted something new, but perhaps the task positively demanded a fresh approach if it were to be achievable. Bear in mind that this took place in the heyday of waterfall development, when companies would proudly announce that they were using it – unlike today, when they try to pretend that they aren't using it by calling it something else. In a climate of big bang integration, lots went wrong when parts were brought together, so although it is absurdly easy to criticise with 20 years of hindsight, the appeal of RTSASD is an understandable one. I must also give the company credit for recognising the need to invest in training and consultancy to embark along this new path. Nonetheless it was a painful experience at times. The system was delivered eventually, but not without endless meetings, estimates, slip, arguments and re-estimates.

Example 2: GDPM

The company must have been in a period of hard reflection on its processes and certainly cannot be accused of being set in its ways at that time, because soon afterwards came another project with a new process – this time for the project management. The project, like many the company undertook, was essentially an update to an existing system for a new customer, and like all radar projects it involved a mix of mechanical, microwave, digital electronic and software engineering. As it was based on an existing system, the myth of code reuse had dictated the estimates for the software; I disagreed with the figures I was presented with at the start of the project and estimated the task at twice the original hours, but was overruled and the schedule was based on the myth. No prizes for guessing how long it actually took, given that my estimate was based on expertise, experience and analysis rather than a simplistic assumption that it couldn't take long because we could just plug in the code we already had and tweak it a bit. This scenario is well described in a paper by Ozarin [2] under the heading 'excessive faith in code reuse'. In the face of such denial nothing could have made the project run to plan, not even Goal Directed Project Management (GDPM).

GDPM seems to have survived the test of time and nowadays has its own website [3], which describes it as 'a straight forward approach to project management' – though not one that extends to much quality control, evidently, if the fact that the website refers to goal direced [sic] project management is any indicator! To summarise very briefly, it defines a series of project milestones and draws up a cross-functional matrix of responsibility for their delivery.

Once again, my employer invested in its decision and paid for training for all staff on the project. Like the RTSASD training, inevitably the examples were simple ones. Unlike the RTSASD trainers, though, the GDPM trainers cheated in order to boost their claims of how much more efficient a GDPM project could be. It's impossible to construct a rigorous double-blind test to examine the results of asking people to do things differently; even if the assessors of the results can be kept ignorant of the method used to achieve them it is by definition impossible for the participants to be unaware of it! And even if we accept this known weakness of our experimental method, it is not possible to be certain of starting from a level playing field. Either two teams are chosen at random and asked to complete the same task with different methods, or the same team is asked to complete multiple tasks with different methods. The former cannot be immune to the huge differences we routinely see between teams and the latter cannot control for the experience gained as the experiment progresses.

So in fact when the GDPM trainers set two Lego construction tasks, one in the morning where we were left to our own devices and one in the afternoon following specified principles, we could have achieved a significant improvement for the afternoon's performance just by what we had learned from the morning's experience. Unless the principles we were required to follow positively hampered progress, things were certain to improve. But the trainers made doubly sure by quietly shifting the rules. Suddenly hugely useful pre-manufactured sub-assemblies became available, this information magically appearing without the need for application of new GDPM skills. Speaking subjectively, the 'suppliers' and 'customers' played by the trainers also seemed to be surprisingly soothed into a more cooperative mood than could be accounted for by their sandwich lunch.

All the management talking up the great new solution they had found had already taken this training. Actually there was good stuff in it, but it's alarming that they didn't seem perturbed by the quality of this 'evidence' for its effectiveness. Good evidence is very hard to find; unsuccessful companies are unlikely to wash their dirty laundry in public if they can possibly avoid it and those who succeed could be doing so for any one of a multitude of reasons, or indeed a complex cocktail of factors. Given these difficulties, perhaps the most reliable evidence to look for when evaluating claims is a logical, coherent and adequately comprehensive strategy. If the vendors come from a respected source with a good track record and show a similar understanding of the problems, what's useful and what's not, so much the better. Sadly some people seem to be satisfied with Lego, smoke and mirrors.

If I am hard on the trainers it is partly because I think they did a promising method an injustice. The most notable result of applying it was better communication, with more opportunity to meet with people from across the project with a cross-section of expertise. Interestingly it managed to achieve this without co-location. Hard to imagine, I know, that this project predated ubiquitous office email and project management software. A high-tech business without email is inconceivable now and the website reports that GDPM is 'supported by a comprehensive tool'; I cannot help but wonder if the main benefit it brought for us has been endangered by progress.

While communication was much improved, it will be obvious from the software estimating story there was no greater success in meeting milestones; GDPM's focus on finding ways of meeting the goals cannot contend with the effects of such defective planning. Nor did it seem to improve the ability to foresee the missing of milestones; perhaps it was too intent on struggling against the odds to meet the goal at all costs. Another factor may be the company's own persistent hierarchical culture. Consultants were used from the start to help implement the method but their contact was almost exclusively with The Management, so while many discussions took place, they did not involve the people who were getting their hands dirty doing the work. The method was not a heavyweight one and showed potential but it was not skilfully implemented, perhaps because the company culture was not ready for it.

Where they go wrong

I hope it is clear, from the fact that I see some benefits among the pitfalls of these examples, that I am not suggesting that methodologies never deliver improvements. And presumably it is normally a perceived need for some improvement of some kind that ultimately motivates their adoption, because even those who seek out new for newness' sake usually have to make their case with some claim for the advantage that can be gained. My contention is simply that adopting a methodology as a solution rarely lives up to expectations – and perhaps not surprisingly since expectations are bound to be high; why would anyone embark on such significant and wide-ranging changes if they were not optimistic about the benefits? The reasons that these benefits fail to be realised fall broadly into three categories: approaches that have poor potential but nonetheless manage to look good; approaches which actually have good potential but are thwarted by poor implementation; muppet teams!

What problem are you trying to solve?

If I were to use one word to describe those in the first category it would be meretricious:

> Alluring by false show; showily or superficially attractive but having in reality no value or integrity. (OED)

Although the word sounds as if its origins may lie in the merit that something lacks but appears to have, in fact it is derived from the Latin word for a prostitute, its earlier meaning being 'befitting a prostitute' (OED). The market for the oldest profession is long established, but new technology has established new markets ripe for exploitation, and not just in internet porn! In a software industry of missed deadlines and massive overspends any promise to counter these is bound to be very seductive. In addition, managing programmers is one of those tasks that has been described as 'like herding cats', so anything which may simplify it is clearly desirable. Having someone else work out how to actually achieve all these difficult things is surely tempting, but as The Pragmatic Programmer so succinctly puts it:

> No matter how well thought out it is, and regardless of which 'best practices' it includes, no method can replace thinking. ([1], p.201)

Many types of person are vulnerable to the appeal of a predetermined formal solution. Budget holders are the most obvious because they are, of course, particularly susceptible to the siren call of cost savings – but they are rarely the people best placed to critically assess the potential for achieving them. Those who actually carry out the work are likely to ask more searching questions, and software developers in particular are perhaps also more likely to have the analytical skills to assess the claims rigorously.

Less obvious but nonetheless susceptible targets are those who are looking for a mast to nail their colours to. We might characterise these as: 'fundamentalists', who seek 'one true way'; 'academics', who find theory very persuasive but perhaps lack the experience of the messy interface where theory meets the real world; and 'seekers', who are always keen to try the next new thing and often find themselves in possession of a solution looking for a problem.

The frightening targets, because of the huge amount of influence they are likely to wield, are the corporate psychopaths or 'snakes in suits' [4]. For a smooth, polished and often charming mastermind of Grand Plans a suitably glossy and high-profile methodology is a gift. It looks good, and if the project succeeds there is no need to analyse what went right to achieve this; clearly it was their foresighted choice of a world-class, best-of-breed solution. If, on the other hand, the results are disappointing – and the corporate psychopath will stoop to very low tactics to avoid this – any potential for blaming the mastermind is much diluted because there is nothing too shameful about having used a respectable method widely recognised by the industry. And having made a sound choice, surely the outcome was not really in their hands; it would have worked were it not for poor execution by subordinates.

Of course there are also those who will be reluctant to accept changes, and they contribute to a cultural explanation of the second category of failures: sound methods that falter in their implementation. In some cases the resistance is a passive one, a fear of change rooted in the lack of confidence to try something new. These people have found a way of working they are comfortable with and do not want to step outside their comfort zone. More active resistance comes from those who are (over) confident – adamant that they know how to do their job perfectly well already and affronted at the idea that anyone could tell them how to do it better. These, by definition, are not the organisation's 'best people'; the best people know there is always more they can learn.

The supremely confident people are likely to rail against the imposition of any measure from above that attempts to tell them how to do their job, while the timid are likely to evade it (for example, by keeping their head below the parapet, working away in a quiet corner on legacy projects). However, top-down rules are not the only way in which power is exercised. The sociologist Michel Foucault described how power circulates in the ideas, values and norms that people internalise. That people are unaware of their self-imposed rules is evident in the way they will describe them as 'natural' or 'obvious', if they think about them at all. The internalised rules prevalent in an organisation silently but powerfully govern behaviour. For example, a former colleague once talked about 'not being allowed to discuss salaries with other employees'. While some companies do have such policies, no such rule existed at the company in question, but the prevailing understandings about appropriate conversation were strong enough to constrain people's behaviour just as much if there were a specific interdiction.

The potency of unwritten and often even unconscious rules governing 'how we do things here' has considerable influence to empower the application of explicit measures that fit with the ways of an organisation and to undermine those that don't. I am not talking here about deliberate sabotage, but simply a habitat in which new processes can either thrive or languish – an outcome that will surely be familiar to many previous delegates who have experienced the post-conference slump of heading home full of ideas that then fall on stony ground. The ways of an organisation are, of course, not set in stone but constantly renegotiated through all the daily activities that reinforce or challenge them. Over time significant change is possible, but it is more likely to shift little by little, with a gradual catching on and following of leads, than in a big bang.

I referred to how well measures fit an organisation, and that in itself is a crucial element of a successful implementation. It requires an honest analysis of the situation and good advice to find the appropriate solution; one size does not fit all. I will discuss the practical implications of this in the next section. Starting from somewhere other than such an analysis also pretty much guarantees that you will lose sight of the goal you set out to achieve; if you don't know where you're going you'll end up somewhere else.

Finally the third category: however good the process, no project is going to succeed with a poor team. Identifying good people and assembling them into an effective team has been a challenge since programming began and is far from solved, so it is another ripe market for exploitation for those who claim to have a solution. As with process there are of course also people out there trying to find genuinely helpful answers to this very difficult question. It is a whole separate topic and not one I will cover here, but it is well worth the effort spent on it because, in contrast to the muppet team, a really good team can deliver in almost any circumstances.

What is really needed?

To help get to where you really want to be there is a single question which is more important than any other:

What problem are you trying to solve?

Unfortunately it is too rarely asked. It is important to solve **YOUR** problem, not **A** problem; prepackaged solutions may start from a different launching point, a bit like the old joke about giving directions: 'I wouldn't start from here'. Actually knowing what your most important problem is requires an honest, 360 degree analysis to avoid simplistic assumptions being made about cause. Compare this with the Cardboard Programmer

(or its counterpart the Rubber Duck in [1]). Stopping to describe a situation to an attentive listener is often enough of itself to rapidly identify the underlying problem without a single word (or quack) from the listener. Once you know the problem you are starting from you can apply appropriate measures – appropriateness is key here. In the Cardboard Programmer situation it isn't uncommon to have first tried various ineffective remedies based on inaccurate assumptions about the cause. It is only when forced to examine the symptoms closely enough to be able to describe them to someone else that the truth, almost magically, emerges. If you can't describe it clearly, chances are that you have not yet properly understood the problem.

In coming to an understanding of the problem there is no such thing as a silly question. Answering the questions and, more importantly, justifying those answers, encourages closer examination which fosters understanding of the reasons and helps to flush out spurious assumptions. And just as no question is silly, nor is any potential solution when it comes to visualising the possible ways forward. Ideas should first be generated without judging them. Past experience is one source – have you ever had a similar problem; how was it tackled; how did that go? – but more creative ideas can also flow more freely if they are not immediately judged. Conversely, having a convenient answer already lined up limits your choices and the chance of a satisfactory solution.

Only once you have a body of ideas to consider is it time to judge (constructively!), picking the best idea and working on it to refine it. Because the other options have not been closed down too early, they too are on the table as a source to cherry pick useful elements from. Finally you have a strategy and can create a plan to employ it. If at all practicable, that should mean testing it on a small scale first. In any case, its success in solving the problem it was designed to tackle needs to be monitored. Any mismatch is feedback rather than failure if you then use it to refine your approach in the light of the extra information you now have. Wise people know this and carry out retrospectives; the less wise hold witch hunts or simply sweep it under the carpet and move on with no lessons learned. There is of course also something to be learned from success, and it is likely be all the more memorable if you follow advice to celebrate it ([5], pp.158-160 and [6], p.227).

Visualising solutions

I don't believe methodologies have any place in 'how to do things better' until you have genuinely determined what problem you are trying to solve. Having done that, though, they have tremendous potential in the process of visualising possible solutions. They offer stories that illustrate how things might be, and in picturing that image of the future it is necessary to put ourselves into the story – much like those children's story books personalised with names of the recipient and their friends as the hero and accomplices – to imagine what it can do for us in our own unique circumstances.

As we read the story, do we nod in recognition as we see what purpose an artefact of the process serves and the value it has for us? If the published version of the story doesn't quite fit our own reality, is it flexible enough to bend to the shape of our world or is it prescriptively rigid and therefore liable to break upon meeting our reality? How much are we persuaded by it? Its persuasiveness depends not only on it being coherent and comprehensive, but also on the extent to which it makes sense empirically – largely a function of how much the author's understanding of the evidence chimes with our own. Simple respect for the author plays a part in influencing our acceptance of their story, and that respect has its roots in holding a broadly similar view of the world.

Fictional stories often demand something back from us if they are to work; suspension of disbelief, for example, or imagination – they say the pictures are better on radio! Methodologies make demands of us too, not only mental ones like faith but also physical ones like resourcing for new roles or tools. Before acquiescing to these we must consider whether they are commensurate for every project affected by the decision. The best solution for one is not necessarily the best for another. While it may be desirable to compromise on a good all-round solution to gain the benefits of consistency and familiarity, this just isn't possible in circumstances where what's 'best' varies enormously. ClearCase, for example, may be an ideal choice for a large multi site project; it requires a certain amount of admin effort but that is an appropriate overhead in return for the benefits. For a small project, however, it could be considered grossly heavyweight and the overhead would be unaffordable.

ClearCase is not, so far as I am aware, a prerequisite for any methodology, but it does demonstrate how a one-size-fits-all approach to tools can have serious implications for a heterogeneous 'all' by having to choose the highest common denominator. But tools can be a problem even if there is a homogeneous 'all', or even a single project. Personally, I'm immediately sceptical and suspicious of the ideas someone is trying to sell me if they also have a tool to sell me to help implement those ideas, but you don't have to be a cynic to have reason to be wary. There are questions which are more important than whether they have a vested interest.

Here are just some of the questions you need to have answers to. Is the tool the master or the servant – are you doing things for a sound business reason or to appease the mighty tool? Is it able to integrate with your existing organisation? An important aspect of this might be the ability to import and export information in a plain text format so that it can be linked in with other systems. Closely related to this is its support for automation; can that export be scripted and scheduled or does someone have to sit at a GUI following a series of manual steps, time-consuming and each prone to human error? Can everyone who needs to use it readily do so? If it's so complicated that only a few experts can use it and act as interlocutors, or it is so expensive that licences are closely rationed, it is a hindrance. If it only runs on a different operating system or a different network to the one you are using for development, forget about it.

A sales pitch is unlikely to answer these questions adequately, though you may well discover enough to rule it out if you have the right people present to probe deeper. One suggestion is to request a complete set of user documentation from the vendor before they are allowed to set foot on the premises, the rationale being that if limited time is available for evaluation it is better to spend all of it on the details than to squander any on installing and setting up a trial version and so spend less time actually looking at how you would use the product. Whichever approach you use, do ask these questions as if it were your own money you're spending.

Tailoring the solution

If visualising how you might adopt a methodology suggests it is a potential solution then it can go forward into the process of selecting a solution and refining it; the other solutions can be raided for useful nuggets. There are purists who baulk at the 'refining', taking an all-or-nothing stance which claims that the benefits will not be realised if you do not do everything by the book. (Incidentally, one of the most vehement purists I have encountered was actually not doing things in strict accordance with his holy book, much as he believed otherwise. The uncertain meanings of the badges people use to describe their approach are a whole separate topic!)

I disagree with the purists; they seem to lose sight of the real goal – solving your problem – and pursue a different one of adherence to method. An existing, coherent set of ideas is almost certainly a good place to start instead of inventing your own, but mindless adherence to them certainly isn't. Critical faculties must be engaged all the time or the tail will wag the dog. This doesn't imply criticism of the method but simply the fact that off-the-shelf solutions are by definition not tailored for a perfect fit:

> It is a principle of methodology that the power of a method is inversely proportional to its generality... To be powerful, a method must exploit the problem's features very minutely. Because problem features vary widely, we need a repertoire of methods, each suitable for problems of a particular class. (Michael Jackson, [7])

Santa Claus appears in the title of this piece because he offers a fine example of a powerful and established story which is never followed literally but always with legion accommodations to local abilities and needs. The goal is to help create a happy Christmas for children (and thus improve the chances of a happy Christmas for adults!) They get a lot of pleasure not only from the presents but from the magical nature of their

delivery. The story is one of elves, flying reindeer, and dropping down chimneys, but those of us who implement it don't take that literally. Bringing the magic to Christmas Eve and Christmas morning is achievable by more pragmatic means that don't involve living at the North Pole or defying the laws of physics. Furthermore, every family tailors elements of the story for themselves. Santa is rumoured to enjoy milk and cookies, but when visiting my house he shared my mother's taste for sherry and mince pies. His generosity seems to be tempered by the family budget. And despite the established place of a chimney in pictures of Santa at work, lack of one is a potential hindrance that has been overcome by many diverse solutions suited to the local terrain. The magic works because we combine the power of the story with our own ingenuity.

So, as Santa shows, one size doesn't fit all, at least not without some alterations. The paradox is that anything too general is unlikely to be of much use, while anything sufficiently specific is unlikely, without refinement, to be entirely appropriate. It takes local knowledge to determine how to 'exploit the problem's features very minutely' – knowledge that is perhaps best gained with the help of an independent consultant who can bring a fresh perspective, because it is hard to see these things objectively from within. Also very helpful in this process are resources such as the Organisational Patterns book [5], which offers a pattern language approach progressing step by step through measures appropriate to the situation that prevails following the previous step.

In the search for a good fit, all sorts of resources can be called upon. Methodologies are a means of circulating ideas and communicating principles and provide a good source of theories, even if they are sometimes akin to what is described in The Science of Discworld [8] as 'lies to children' – explanations of complex phenomena simplified to help people understand. Like the description of the atom as a miniature solar system, they are a starting point from which to build a more subtle understanding.

People in other fields make successful use of this integrative strategy:

> The mind is something you have to gain control over. I spent a lot of time on this ... and I have read massively into it. Some of my reading was business-related, about the principles of successful living, some of it is understanding the brain and how it works and quite a lot has been more philosophical. I have, for example, learnt a lot from the doctrines of Buddhism. Don't get me wrong. Don't think: 'Jonny's now a Buddhist.' I am not. I have just been finding a direction, learning different ways of looking at life and taking bits I could use and discarding bits I could not.
>
> *(Jonny Wilkinson, [9])*

And in a respectable pragmatic approach we could learn from, many psychotherapists, rather than being, for example, strict Freudians, integrate therapies from a variety of different theoretical backgrounds into their practice and can call upon whichever is the most appropriate for a particular client at any given stage.

Conclusion

When it comes down to it, the team is the most important factor.

> There is some resistance in our industry to the idea that people factors dominate in software development.
>
> ... I kept discovering that successful teams were still delivering software without using our latest energy-saving ideas.
>
> Initially, I viewed this as a nuisance.
>
> Eventually, it went from a nuisance to a curiosity ... the opposite correlation held: Purely people factors project trajectories quite well, overriding choice of process or technology.
>
> I found no interesting correlation ... among processes, languages or tools and project success.
>
> *(Alistair Cockburn, [10])*

Obviously this doesn't imply we should just forget about processes and concentrate on getting together a good team; they are hard to find! The perfect process is even more elusive... because it doesn't exist. The value of a process depends on its applicability to the problem and the current circumstances, which are ever changing, so it must be flexible enough – and subjected to enough intelligent thought – to cater for a dynamic arena. Santa has survived the disappearance of chimneys from the average family home; an intelligent approach to process can be equally durable. ■

References

[1] Hunt, A. & Thomas, D. (1999) *The Pragmatic Programmer*, Addison-Wesley, ISBN 020161622X

[2] Ozarin, N. (2007) 'Lessons Learned on Five Large-Scale System Developments' in *2007 1st Annual IEEE Systems Conference*, IEEE, ISBN 1-4244-1041-X

[3] www.gdpm.com (accessed 30/11/08)

[4] Spinney, L. (2004) 'Snakes in Suits', *New Scientist*, 21/08/04

[5] Coplien, J. & Harrison, N. (2005) *Organizational Patterns of Agile Software Development*, Pearson Prentice Hall, ISBN 0-13-14670-9

[6] Kelly, A. (2008) *Changing Software Development: Learning to Become Agile*, Wiley, ISBN 978-0-470-51504-4

[7] Jackson, M. (1995) 'Problems and Requirements' in *Proceedings of the Second IEEE International Symposium on Requirements Engineering*, IEEE, ISBN 0-8186-7017-7

[8] Pratchett, T,. Stewart, I. & Cohen, C. (1999) *The Science of Discworld*, Ebury Press, ISBN 0 091 86515 8

[9] The Times (2007) http://www.timesonline.co.uk/tol/sport/columnists/jonny_wilkinson/article2697308.ece (accessed 30/11/08)

[10] Cockburn, A. (2002) *Agile Software Development*, Addison-Wesley, ISBN 0-201-69969-9

Trouble With Dates

Mark Easterbrook takes us on a date.

Think back to the first of January 2000 CE. The world had not stopped. A last minute effort by the world's computer programmers had averted global meltdown. Everyone could relax until early in 2038 because software developers had learnt all about dates, albeit the hard way.

Unfortunately, history, as always, complicates matters, so there are plenty of things to consider long before the next significant date rollover. In this article I am going to look at a number of date problems and what caused them, whether you need to do anything, and if so, what.

The leap year problem

Before Y2K, a common technical interview question was the calculation of a leap year. A typical manifest was to provide a sample line of code and ask what is wrong:

```
is_leap = year%4 == 0;
```

Sometimes no code is provided and the candidate is asked to suggest an implementation.

The intent of the question is manyfold, such as:

- Write an expression in a clear and simple way.
- Understand operators, especially the modulus operator and operator precedence.
- Initiate a discussion on leap year calculation, the Y2K problem, and other issues that can arise from taking a too-simplistic view of the problem domain.
- Discuss the design of a date solution to demonstrate ability to design. This might include date and datespan classes.

Adjusting the test for century years, such as 1900, which are not leap years, results in:

```
is_leap = year%4 == 0 && year%100 != 0;
```

and introduces one of the Y2K problems, because 2000 is a leap year.

Other 'improvements' such as:

```
is_leap = (year%4 == 0);
is_leap = (year%4) == 0;
```

show a lack of knowledge of leap year calculation and naivety of operator precedence.

```
isleap = year%4==0 && (
   year%100!=0 || year%400==0);
```

is an overcomplicated solution, vulnerable to typing errors, and wrong. I bet most of you are googling at this point, and you will find the most common 'answer' is the same as above, but it is still wrong.

A correct answer for the majority of applications is:

```
assert(year>=1918 && year<=2099);
is_leap = year%4 == 0;
```

If you understand why this is a 'good' answer, you will also understand the following joke:

Q. In what month did the (Russian) October Revolution occur?

A. November.

MARK EASTERBROOK

Mark has experience developing embedded systems, OSes, telecommunications systems, and command and control. When not in front of a computer he rides motorcycles and horses. Contact him at mark@easterbrook.org.uk

Predicting the future

Before I look at history, let us take a quick look into the future and why I choose an upper limit of 2099. Am I not introducing a year 2100 problem? I might be, but I don't care, and neither should you.

1. The chances of the code still being used at a time that it actually matters is very small. Even if you are writing code for 40-year life insurance policies, it is not a problem until the late 2050s; most other applications won't care until the 2090s.
2. A lot of date handling code will crash and burn before then anyway, mostly in 2038. This will focus attention on year 2100 issue and everyone will have 60 years to prepare, which is about 59 years longer than most people spent preparing for Y2K.
3. We don't actually know if 2100 is a leap year or not. It will be a leap year if we continue to use the same calendar with the same definition of a leap year that we use today – but a lot can happen in 90 years, and our current calendar has only been universally used for 90 years.

I hope we can agree that trying to support dates beyond 2099 (where the leap year matters) is a pointless waste of time.

What the Romans did for us

To be able to support dates prior to 1918, it is necessary to understand how we arrived at today's calendar.

Roman dictator-for-life Julius Caesar, concerned that the calendar was not keeping aligned with the seasons, introduced in 45BC the Julian Calendar with a year of 365¼ days, with the ¼ day handled by adding an extra day every 4 years. This was a much more accurate calendar than those that preceded it, but still 11 minutes longer than a real earth year.

The 11 minutes per year accumulated, but although it was known from as early as the 2nd century CE, it was not until the mid-1570s, when it was 10 days behind the real seasons, that anyone did anything about it. The spring equinox was falling around the 12th of March resulting in Easter occurring too late in the real spring time. Pope Gregory XIII put his best scientists on the case, and so by Papal Decree, the 4th October 1582 was followed by the 15th, thus deleting 10 days; in future 3 leap years in every 400 would also be eliminated (the 'divisible by 100 but not 400' rule).

And thus the Gregorian calendar we use today come into being. Except that even in Catholic lands only Italy, Spain, and Portugal switched on time. Everyone's birthday moved to a calendar date 10 days later, and rents, interest and wages had to be recalculated for a month that had a mere 21 days (and you thought the Y2K problem was a difficult one).

France switched in December, parts of the low countries jumped from 21st December 1582 directly to 1st January 1583, skipping Christmas. Most of Catholic Europe had switched by 1584, but as at this time Europe was a patchwork of feudal states, travelling across a border could mean the calendar moving backwards or forwards 10 days in time. Think about this next time you have to consider today's timezones.

1752 and all that...

Resistance was strongest in Protestant and Eastern Orthodox lands. The Julian calendar was followed until 1752 in Britain and its colonies, and it wasn't until after the October Revolution in Russia that the Gregorian calendar became universal.

The 1752 change in Britain was an opportunity to make another adjustment: moving New Year's Day from the 25th March to the 1st January; so in fact 1752 was a very short year, running from the 25th

March, skipping 10 days in October, and finishing on the 31st December. This was unacceptable for some annual contracts and financial dealings, so the accountants of the day kept their year starting on the Julian 25th March, which is now, after leap year adjustment, the 6th April in the Gregorian calendar.

Handling dates before 1918

It should be clear by now that if you need to handle dates prior to 1918 it is not sufficient to just validate and store a modern style date. Unless your application is unique to a country or area, some, or all, of the following will be needed:

- The date converted to the Gregorian calendar CE/BCE extrapolated backward if necessary. This allows date span calculations.
- The original date including calendar and country/origin. Probably free text to cater for historical methods of recording dates such as years relative to the monarch reign.
- Handle fractional year notation (e.g. 1751/2) for dates between 1st January and 25th March.
- A method of converting between calendars.

More up-to-date problems (no pun intended)

Even post-1918 the Gregorian calendar is not a given, with the Jewish and Chinese as examples of calendars in daily use. However, if you choose to only support the Gregorian calendar, there are still plenty of ways to design-in bugs.

Day-month-year order and notation

There are a number of ways to order dates:

- The USA uses M-D-Y order
- Some Scandinavian and Far East countries use Y-M-D order
- The international standard for dates (ISO8601) also uses Y-M-D
- Most of the rest of the world, including most of Europe, uses D-M-Y
- Military notation is D-time-M-Y order
- Unix influenced systems use Weekday M D time timezone Y order
- Email dates are Weekday, D M Y, time, timezone

Dates are usually delimited by a slash or hyphen in the English-speaking world, and dots in Europe.

Parsing and displaying all these various formats is a difficult but solvable problem, except for the ambiguous difference between the US M D Y order and the D M Y order. On a single machine this can usually be solved by reference to the locale settings, but in a network architecture, especially a web based solution, the locale of the server may not be that of the user, and browsers do not have a locale accessible by the server. Various methods have been used to solve this issue:

- Use the month name instead of the number; however, this may introduce a language problem.
- Drop down lists for input so the month field is constrained to 1-12 so it is obvious to the user. This does not eliminate the problem, it just makes it less likely.
- Display a pop-up calendar. It is quite tricky to ensure that this works in all browsers and configurations.

The issue is further complicated by US written software referring to the M D Y order as 'English' format.

The Excel 1900/1904 problem

Dates in MS Excel are represented by the (floating point) number of days since an epoch. Time is encoded in the decimal part. There are two quirks to the Excel implementation that can cause no end of headaches:

1. The epoch chosen for DOS/Windows is 1900, and for the Macintosh is 1904. Thus anything that interfaces to internally represented dates stored in Excel needs to check which epoch is in use.
2. The representation treats 1900 as a leap year so that serial value 59 is 28th Feb, 60 is an invalid date, and 61 is 1st March. This means that for dates before 1st March 1900 the `WEEKDAY` function returns the wrong value and dates cannot simply be subtracted to obtain the number of days between two dates.

This has become a topical issue recently as both of these anomalies have been ported unchanged into the OOXML specification. The ECMA document actually forbids applications from supporting years before 1900!

The day, the whole day, and nothing but the day

The most common internal representation is to encode both date and time in a single value, for example, Unix uses the number of seconds since 1970 and Microsoft uses the fractional part of the value as time of day. Although this makes it easy to calculate the elapsed time between two values, it breaks down when only the date is required. It is common to set the hidden time component to zero (00:00 or midnight).

If you have a portable device containing a calendar, try setting a date-only event such as a birthday (often called an all-day event). Now travel to a far away timezone, set your device to local time, and look up that date event. If it now runs from sometime on one day to the same time next day, that hidden time has bitten. That date is only really midnight one day to midnight the next in your local time zone.

You don't even have to travel halfway round the world. Just wait for one of the twice annual clock changes known as daylight saving. Are you sure your carefully recorded date with the invisible 00:00 time doesn't magically become 23:00 the previous day?

Choosing 12:00 instead of 00:00 avoids some of the issues, but do you really want code for 'a day plus a half' all the time?

The year 2038 problem

This is probably the most well known of the future date problems (second only to Y2K if the past is also considered). In Unix based systems, date and time is stored as the number of seconds since an epoch of 1st January 1970. This will overflow a 32 bit signed integer in the early hours of the 19th January 2038 and suddenly the world will be plunged back to sometime December 1901.

Of course the naive answer is just to recompile your application for a 64 bit architecture because by then there will be no 32-bit machines still running. This is not a solution because:

1. Today while we are transitioning from 32- to 64-bit architecture, there are still a lot of 8-bit processors in the world, probably more than the total 32- and 64-bit combined. There are likely to be 8-bit systems still running in 2038, and certainly plenty of 32-bit.
2. The 2038 problem only occurs in 2038 for the real-time clock. For anything looking forward it will occur before then. In May 2006 some server software crashed because it used one billion seconds in the future to mean forever. Twenty-five years is a common term for mortgages and life insurance, and thus are only 5 years away from 2038 rollover. It is all downhill from there.
3. Recompiling 32-bit applications for 64-bits is often non-trivial, even if the original programmer was knowledgeable about portability issues.
4. Where time is part of a binary file format, recompiling won't work, and if all components cannot be upgraded together, interoperability between 32- and 64-bit parts is a problem with no easy solution.

As we approached the year 2000, it was mostly old Cobol and database systems that were affected. The number of processors and our dependency on them has mushroomed since then and this trend is likely to continue, meaning that solving year 2038 issues is going to be more important than solving the Y2K ones was.

Exception Handling in C++

Andy Farlow demystifies C++ exceptions.

If you have not used the C++ exception handling mechanism before, it may seem a little daunting. It's one of those language features we may decide is optional and therefore safe to ignore.

But here are two points to bear in mind: firstly, C++ exception handling really is quite a simple concept (OK, like much of C++, it can be found to be complicated if you dig deeper, but there is no reason to dig too deep too soon) and secondly, if you're using code that someone else has written which throws exceptions, there's the distinct possibility that your application is going to terminate unexpectedly at some point with an 'unhandled exception' error. And depending on your application's target audience, that may be unacceptable.

Exceptions versus error codes

Essentially, exceptions are a very good alternative to using error codes. When you use functions that return error codes, what can happen is that the expected (successful) run-time flow of a program becomes mixed up with the exceptional situation run-time flow. You find that in higher level functions you have to concentrate on handling error situations that were caused in lower level functions. The normal run-time flow becomes littered with error processing – and these error codes often have to be passed up through several levels of function calls.

It's important at this point to make the distinction between expected situations and exceptional situations. Expected situations are concerned with all the possible events which can occur as a result of the software working as you have designed it. You may well have designed your code to deal with the user doing something silly (for instance sending a blank email). So your code may check the body of the email and display a warning message to the user if appropriate. This should be considered as expected run-time processing. In this situation, an empty email doesn't represent an exceptional situation. It's rather just one predictable situation that may occur which requires handing in some way.

Listing 1

```
void someClass::highLevelFunction()
{

  try
  {
    lowLevelFunction();
    cout << "lowLevelFunction worked OK";
  }

  catch (…)
  {
    cout << "Warning : lowLevelFunction Failed";
    return;
  }
  // Do some more processing
}
```

But, say that in trying to send an email, the piece of low level code that opens up the network connection finds that it has run out of system resources and it can't actually send the email. What then? It could return an error code, and all the function calls between this low-level function and your high level calling code could propagate this error up for you to deal with. Or, this low level code could 'throw an exception' that your higher level code simply 'catches' and deals with outside of its normal processing. This is the benefit of exception handling.

ANDY FARLOW

Andy's been a developer for twenty years – long enough to realise that 80% of software development is just maintenance. He holds that code is only ever clever and fit for purpose if it's easily modified when that purpose changes. He likes trees. Contact Andy at: andy.farlow@ntlworld.com

Trouble with dates (continued)

Leap seconds

Leap seconds are not strictly a date problem, but they can have an effect on date calculations if sub-second accuracy is important to you. As our planet wobbles around our sun, slight differences between astronomical time (time relative to the stars) and UTC (time recorded by atomic clocks) occur. For those for which this really matters, leap seconds are inserted into (or deleted from) UTC to keep them in line.

This means that the length of a UTC year can vary by 1 or 2 seconds, and cannot be predicted in advance, which makes any time representation that is the number of seconds from an epoch rather difficult to maintain and calculate. The solution for Unix time is to pretend leap seconds don't exist.

If alignment between Unix time and UTC is important, you will need code for the anomaly every time a leap second occurs, for example:

- 2008.12.31 23:59:59 UTC will be followed by 2008.12.31 23:59:60 UTC.
- 2008 12.31.23:59:59 Unix time will be followed by 2009.01.01 00:00:00.

Which means no matter how close you can synchronise the computer clock to UTC, there will be a second where they represent different dates, and any UTC time feed may validly contain 60 in the seconds field.

Finally…

Some of you will be lucky enough not to have to deal with dates, and this article will have been simply of academic interest. The rest of us will have to deal with at least some of the problems with dates discussed, the most unlucky with all of them, although hopefully not all on the same project!

Understanding the anomalies and history of calendars allows the prudent software developer to know how much effort to put into handling dates, and especially to know what can safely be ignored for the problem domain under consideration.

And of course you now know that someone with a Russian birth certificate for 1 April 1908 won't be celebrating his or her 100th birthday on April Fools' Day this year. I'll leave it as an exercise to the reader to work out when the party will actually be. Don't be late, or early! ■

Listing 2

```
void SomeClass::lowLevelFunction()
{
  bool bResourceAllocationFailed = false;
  // Allocate some system resource
  bResourceAllocationFailed =
     getSomeSystemResource();
  if (bResourceAllocationFailed)
  {
    throw std::exception;
  }
  else
  {
    // Do some more processing …
  }
  Return;
}
```

So how does it work?

Like many concepts, it is best illustrated with an example. In a high level function you may have some code as shown in Listing 1.

This code segment essentially says: try to call **`lowLevelFunction()`**, but if that function fails in some exceptional way, then we expect that it may throw an exception, so we'll catch it. Once caught, we'll display a warning and return. If **`lowLevelFunction()`** works as expected, then we'll just carry on and do some more processing.

The called function may look something like Listing 2.

In **`lowLevelFunction()`** as soon as the **`throw`** statement is executed, control leaves the function and goes up to the **`catch`** statement in **`highLevelFunction()`**. If the **`throw`** statement is not called (because we were able to get the system resources) then execution just carries on and the code goes on to do some more processing before the function returns.

Any piece of code may throw an exception. As we've seen above, it just specifies something like:

```
throw <exceptionObject>
```

where **`<exceptionObject>`** is any valid C++ object. It could be any built-in type (e.g. **`int`**, **`float`**, **`bool`**) or a class instance. Most commonly a class instance is used, and the class itself will be designed to represent something about the exceptional situation. For example the class may include a method which returns a string indicating what has gone wrong. In this way the calling code doesn't have to do any specific processing (the nature of the exception is encapsulated inside the exception object).

In the example above, we just used the standard library exception class **`std::exception`** (often you'll use something derived from this class).

It's also possible to just **`catch`** an exception and to re**`throw`** it. In this case, you simply use the statement **`throw`** on its own (you can only do this within a **`catch`** block). This in effect passes the exception on. More will follow on this shortly.

Catching

Catching is achieved using a **`catch`** block which is just a **`catch`** statement followed by a code block:

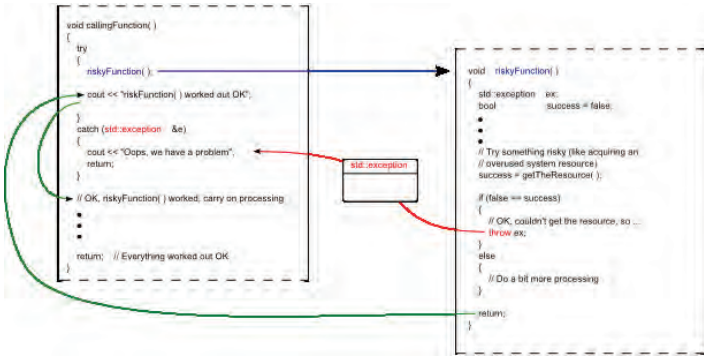
```
catch(…)
{
  // Process the exception
}
```

or

```
catch(<exceptionObject> &exceptionInstance)
{
  // Process the exception
}
```

The first form says catch absolutely any object that is thrown. Processing of that object is impossible because there's no way of knowing what it is,

Figure 1



but at least the program can recognise that something has gone wrong (rather than just exit, that is, crash).

The second form says catch any exceptions of the type specified by **`<exceptionObject>`** and in the following code block that object may be used, if necessary, to process the exceptional circumstance.

If there's no code to catch a particular thrown exception, then the program will terminate. But, the whole purpose of exceptions is that they should be caught – and when used to their best advantage, they're caught by code that's higher up the call stack from the function that throws the exception. Although there's nothing to stop you writing code that throws and catches an exception in the same function, it is just a little pointless – the reason why should become clearer below.

The code that catches the exception can then use the exception object in some appropriate way to process the exceptional situation. Useful processing is often to report the error in some way (i.e. notify the user or write a suitable message to an error log). Or, it can ignore the exact nature of the error and do something else that the developer deems appropriate. The important point is that the program won't terminate unexpectedly and that it deals with the exception as gracefully as possible.

It's important to remember that the code in the catch block will only be executed if some code within the try block threw an exception – and that includes any code that's in any functions that are called – and that means called right down the call stack. If no exceptions are thrown, all code within the try block is executed then execution continues on the line following the end of the catch block. This is all shown in Figure 1 (simple exception handling).

In Figure 1, you can see the alternate paths taken if **`riskyfunction()`** works (the long arrow from its return statement) or fails (the short arrow via the exception box).

Some more details

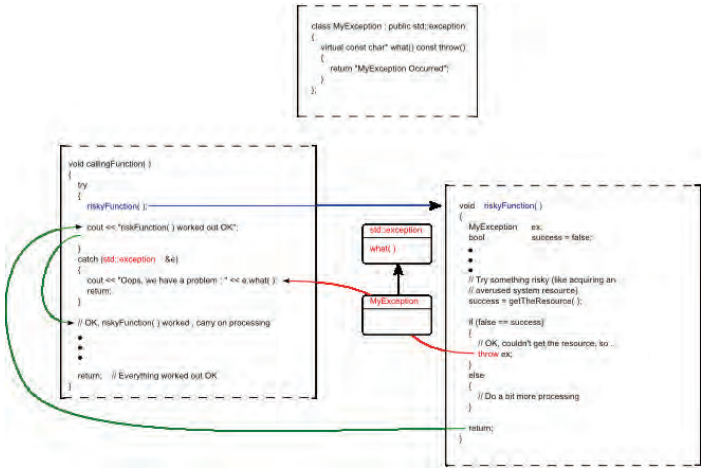
Every catch block has to be preceded by a try block. The point is that your code does a try on something that should work, but it's ready to catch exceptions when they arise.

Notice in Figure 1 that the code on the left explicitly catches an object of type **`std::exception`** (or rather a reference to one – more on this later). In this case it will catch all instances of **`std::exception`** and importantly, any instances of classes derived from **`std::exception`**. This is shown in Figure 2 (more useful exception handling). However, it won't catch exceptions of other types.

Stack unwinding

An important point about exceptions is that as soon as one is thrown in a called function, then all local objects within that function are automatically deleted and cleaned up by the run-time environment (this magic occurs because the compiler adds the code to do it for you). This is significant because it means that all the relevant destructors are called on those objects and (if the classes for those objects are written properly) any resources allocated therein are tidied up automatically. The result is that the code that catches the exception doesn't have to do any cleaning up and de-allocating of resources.

Figure 2



Furthermore, if you have a function that throws an exception from way down the call stack, then that stack is 'unwound' from the point of the throw right up to where it's caught, and all local objects are cleaned up as it happens. This is demonstrated in Figure 3 (stack unwinding).

Another important point shown up in Figure 3 is that a **`try`** block may be followed by multiple **`catch`** blocks, each with a different type. This means that when an exception is thrown and the stack is unwound, the **`catch`** that matches the type of object thrown is executed.

It's generally considered bad practice to have a try block with many different catch blocks following it. The reason being that this method of design simply mimics the use of error codes and **`case`** statements (or multiple **`if-else`** statements if you'd prefer) to process them. That is, each case clause (or **`if-else`**) would correspond to one of the **`catch`** blocks in determining the program execution path. This is not what exceptions were designed to do.

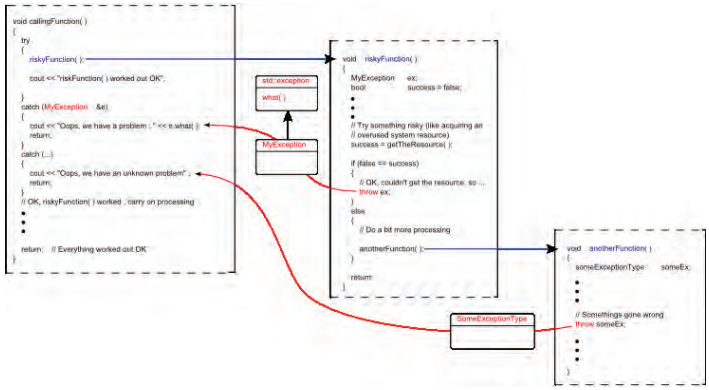
The theory behind it is this: your program naturally will have many different execution paths depending on what inputs it receives. But when an exception is thrown, it really does represent an exceptional circumstance. It's not one that you've designed into the code. So it should ideally lead to just one particular exceptional path (not a choice of many). However this is not a hard and fast rule when it comes to exception handling, but should be thought of as a design goal.

Which catch does the catching

When an exception is thrown, the stack is unwound as explained above, the first **`catch`** statement that is encountered that matches the type of object thrown – or the first one that is declared as **`catch (…)`** – does the catch. If there is no **`catch`** with a matching type and none with **`(…)`**, then it goes all the way up to the top and terminates the program.

As you can see in the example in Figure 3, you can specify that a **`catch`** statement catches a base class so that it also catches instances of derived exception class objects (though to do this they must be caught by reference which is discussed next).

Figure 3



What to throw and what to catch

Generally, the simple advice is to write code which throws temporary instances of classes derived from **`std::exception`** (or some other base exception class, maybe from a specific library you're using, or one you've written yourself). Although, as mentioned before, you can throw **`int`**s, **`float`**s and even **`char`** arrays, it's best to stick to objects (since they may encapsulate the nature of the exceptional circumstance). In all the examples above, you can see that a local exception object is declared in the called function and it's that which is thrown.

When it comes to catching, there's very little reason for doing anything other than catching by reference. In the above examples you can see that the catch is declared as:

```
catch (std::exception  &e)
```

In simple terms then, this code catches a reference to the object that was thrown. If you **`catch`** instead by value (e.g. by specifying **`catch (std::exception  e)`**) this causes a copy to be made of the thrown object (as it would if this syntax were used in a function argument definition for example). If nothing else, catching by value introduces the usual performance hit of additional copying and the inherent copy constructor error potential. If you don't want your **`catch`** block to modify the exception object then just declare it **`const`**:

```
catch (const std::exception  &e)
```

Furthermore, by catching by reference, the code is polymorphically able to catch instances of classes derived from the one specified in the catch statement. This is obviously important if you have a hierarchy of exception classes.

It is almost never a good idea to catch by pointer. For instance, how do you know where the pointed-to object was allocated (stack or heap)? How can you know whether it's your code's responsibility to delete it if it was on the heap? There are other considerations too, so the general advice is just don't do it unless you have a very good reason to do so.

Some more about throwing Exceptions

You may see exception handling code such as:

```
try
{
  someCall( );
}
catch (OneTypeOfException &exOriginal)
{
  AnotherTypeOfException exNew;
  throw exNew;
}
```

This could be considered perfectly valid from a coding point of view and you may see it in existing code or even attempt to imitate it in your own code. You may have very good reasons for doing so. But if you do, you should ask yourself whether or not you're catching the exception in the right place to start with.

You might also want to rethrow the current exception, (possibly with better reasons than you would for the above). The following code illustrates this:

```
try
{
  someCall( );
}
catch (AnException &ex)
// Note the catch by reference
{
  if ( powerHasBeenCut( ) )
  {
    ex.addDescription("Power has been cut!");
    throw; // This rethrows object ex
  }
}
```

Note the fact that throw is used in its own (without an object). This is an example of catching an exception at one point in the call stack, enhancing

its usefulness and passing it on – rethrowing it. This does of course rely on the exception being caught by reference, otherwise the code would just be adding the description to a copy of the current object and when the rethrow occurs, the copy goes out of scope and the original (without the enhancement) is passed on.

Constructors and destructors

Since constructors cannot return error or status codes, it's considered reasonable for them to throw exceptions. The alternative is for them to set some internal state (e.g. **`bool bErrorConstructingObject`**) and for that to be queried by the code that uses the object. By having a constructor throw an exception however, we can do something like the following in the calling code:

```
SomeClass *pSomeClassInstance = 0;
try
{
  pSomeClassInstance = new SomeClass;
}
catch(…)
{
  // Report the allocation failure maybe
  return;
}
// Carry on
```

As long as **`SomeClass`** is written well, this can be a useful way of designing code.

With destructors, however, it's a different matter. The problem lies in the fact that objects may well be under the process of destruction during the stack unwinding process (after an exception is thrown). This, after all, is one of the benefits of the mechanism. So consider the situation where an exception is thrown and consequently is in the process of being caught up the stack somewhere. As a result, a destructor on some object along the way is called. What if that object itself throws an exception. Does the run-time keep unwinding to the original catch or look for a new one based on this new exception? Well, the answer is that it does neither of these things. Rather it's so disgusted by the dilemma that it just calls **`terminate()`** and exists the program (as per the language specification). This is the very circumstance that you were probably trying to avoid with exception handling in the first place. So, don't throw exceptions in destructors unless you've got some very clever mechanism in place to avoid this situation.

A warning on exception specifications

A function may be declared to throw an exception using an exception specification. With these, you simply add the types of exception objects that may be thrown to the end of the function declaration like so:

```
void SomeClass::someFunc(int dArg) throw (int)
void SomeClass::anotherFunc(int dArg) throw (
   int, myEx)
void SomeClass::someAdditionalFunction(
   int dArg1) throw ()
```

The first example declares a function that says that it may throw an **`int`** exception, the second says that it may throw an **`int`** exception and a **`myEx`** exception. The third says that it's not going to throw any exceptions at all. Many functions are declared in this way and you'll see them as such in various libraries and other bits of code.

The trouble with exception specifications is that they look like absolute guarantees that the function will only throw the exceptions that are specified, or that they will throw none at all. On the face of it that makes any code that uses the function easy to write because you only have to catch those types of exceptions (or none at all).

Unfortunately, this isn't the case. In reality your exception specification-limited function can throw other types of exceptions as well. This is because any functions that it calls may throw exceptions – and your function can't act as a guarantor for their actions.

OK, you may say, that's fine. I'll just make sure that my function doesn't call any other functions that throw exceptions, just ones that do not throw anything. In this way, it can keep its guarantee and any code that uses that function can just handle the specified exceptions. They are two points to bear in mind here though. Firstly, if your function does call another function (which may call another function, which may call another, etc) it is unlikely that you can ever have any idea about the exceptions that may be thrown deep down this nested level of calls. The only real way to know what exceptions may be throw is to look at source code – and very often that's either impossible (because you're using a pre-built library) or just way too time consuming and error-prone (not least because someone may change that code in the future – thus breaking your function's guarantee without you knowing it). And that brings us on to the second point – someone may modify your function in the future in such a way that it's able to throw some new exception that's not in the exception specification – again breaking its supposed guarantee to code that uses it.

If you're still not persuaded, then consider this: if your function with its exception specification does actually throw something that is not specified in its specification (despite your best efforts to stop it), then your program will terminate. And it will do this because the C++ specification says (somewhat ironically) that it must throw **`std::unexpected()`** exceptions at run-time under these circumstances – and that terminates programs.

using the exception handling mechanism can add about a five percent performance penalty to your code

Ok, you may say after further consideration, I'll just make sure that my function with its exception specification catches every possible exception (and that's easy, I'll just use **`catch(…)`**) and I'll re-throw an exception that is in the specification. But this rather misses the point of exception handling in the first place. You're just using the mechanism to do the same thing as passing error codes up the stack, rather than using the exception handling mechanism to do the stack unwinding for you (and doing your error processing in just one place).

It seems that most experts agree that exception specifications seemed like a good idea at the time of writing the language specification, but that they turned out to cause more problems than they solved.

One more thing to consider

It's been estimated that, on average, using the exception handling mechanism can add about a five percent performance penalty to your code. After all, the run-time has to do all this stack-unwinding for you. So you may say this is unacceptable and that it's better to use error codes to deal with exceptional circumstances and avoid this penalty.

Remember though that exceptions are designed to handle exceptional circumstances, not to implement the designed (and therefore expected) flows that your program may go through. So, if all is well in your application (and most, if not all of the time, it will be), then this overhead becomes irrelevant. If exceptional circumstances do arise then performance is unlikely to be the program's main problem.

It is also generally considered true that having (unplanned) error handling code intertwined with the normal flow in your code complicates its maintenance. Many of us have had to maintain previously written code with vast and complicated nested **`if-else`**s that deal with really unlikely circumstances and have found that these are perfect nesting sites for all manner of bugs.

Using the exception handling mechanism allows us to remove these bug nests and deal with the circumstances they're designed to handle in much more suitable places in the application. ■

Developer Categorization of Data Structure Fields

Derek Jones investigates how developers create data types.

Data structures are an important aspect of software engineering and while a great deal of effort has been expended on analysing them from a mathematical point of view, almost no effort has been put into analysing the thought processes used by developers when creating data structures.

This article investigates the decision making process behind developers' creation of datatype definitions. Two sources of data are analysed: measurements of existing code (in particular patterns of field ordering within one or more aggregate types, e.g. C `struct` types) and the results of an experiment carried out at the 2005 and 2008 ACCU conferences (this asked subjects to create one or more data structures to handle a specified collection of data items).

This first part investigates various patterns that occur in the definitions contained in a large body of existing C source code, while a second part discusses the results of an experiment that asked developers to create a set of definitions from a specification.

An understanding of the thought processes used by developers when writing code is essential to the creation of support tools (e.g. refactoring or flagging suspicious usage) and coding guidelines.

From the human point of view the organization of data structures is a classification problem. People actively use classification to infer characteristics of objects they have not encountered before. For instance, if I encounter a four-legged animal that barks and has a tail I am likely to classify it as a dog and I can use my existing knowledge of objects that I have previously placed in the dog category to infer some of the likely characteristics of the animal I have just encountered. This process is not perfect, for instance during World War II some of the children evacuated from the built-up areas of London to the country had not seen sheep before and initially classified them as dogs.

Children as young as four have been found to use categorization to direct the inferences they make [1], and many different studies have shown that people have an innate desire to create and use categories (they have also been found to be sensitive to the costs and benefits of using categories [2]).

Analysing the factors that influence how developers organise information into one or more aggregate datatypes requires a more global perspective than can be obtained from measuring source code (e.g. background data on the application domain and program design constraints).

This first part of the article analyses existing C source code, concentrating on finding common usage patterns within individual definitions. Experience with source code suggests that the following are some of the patterns that occur in the ordering of fields within aggregate definitions:

- Fields having the same type will be placed next to other fields having the same type.
- Fields are likely to be placed close to other fields containing information with which they share semantic associations.
- The ordering of fields will follow the order in which information appears in any written specification used as the basis for creating a structure definition.

The second part of the article uses the experimental results to analyse relationships between information that the designer perceives in the application (e.g. in a geospatial application it is likely that the various kinds of location information will be a strong candidate for being grouped together). Any housekeeping information internal to the representation of information within a program (e.g. the `next` field in a linked list) will also be analysed.

The experiment performed at the two conferences asked subjects to create data structures to represent various kinds of information. The results of these two experiments are discussed in the second part of this article.

Analysis of existing source

The following subsection discusses measurements of `struct` definition field usage in the translated form of a number of large C programs (e.g., gcc, idsoftware, linux, netscape, openafs, openMotif and postgresql).

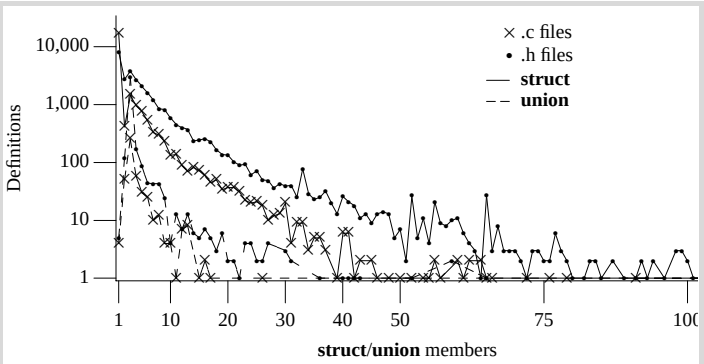


Figure 1

Number of structure and union type definitions containing a given number of members (members in any nested definitions are not included in the count of members of the outer definition). Based on the visible form of the `.c` and `.h` files.

The contents of any header files were only counted once. This C source yielded 6,244 `struct` definitions containing a total of 47,554 fields (average 6.7 per definition).

The following patterns of behaviour were investigated:

- Field type sequences, in particular the observation that fields having the same type are often placed next to each other.
- Shared field names. The extent to which an aggregate definition contains one or more fields whose names also appears in other aggregate definitions.
- Character sequences that are shared between fields in the same aggregate definition, e.g., `farm_house` and `farm_animal` both contain `farm`.
- Influence of specification on field ordering. When a written specification exists, do developers follow the ordering in which it presents information when ordering fields within a structure definition?

DEREK JONES

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

Field type sequences

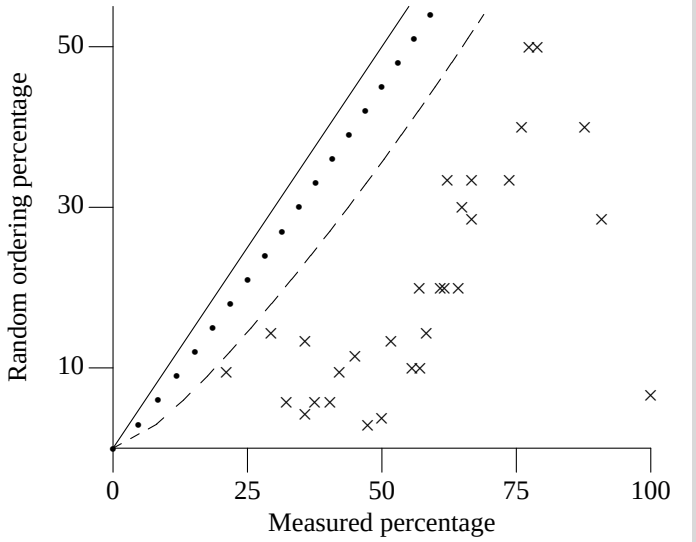
Experience shows that fields having the same type are often placed next to each other within a structure. This subsection investigates this observation.

If, within an aggregate definition, fields of the same type are always placed next to each other, then a definition containing *T* different field types would have *T-1* changes of type. If developers always attempt to place fields of different type next to each other, the number of changes of type has a possible maximum value equal to the number of fields present minus one (this pattern requires sufficient fields of different type that it is possible to continually change type).

The possible sequences of field type within a definition will depend on the number of occurrences of each type. For instance, a definition containing four fields with two fields sharing one type and the other two fields sharing a different type (say **x** and **y**) can have one of six possible type sequences: **xxyy xyxy yxxy yxyx xyyx yyxx** of which 33.3% have the minimum number of changes of type. When two fields share a single type and each of the other two fields have different types there are 12 possible field type sequences: **xxyz xxzy xyxz xyzx xzxy xzyx yxxz yxzx yzxx zxyx zxxy zyxx** and again 50% of the sequences have the minimum number of changes of type.

As the number of fields increases the number of possible type sequences increases. In the case of five fields with three sharing the same type and two sharing a different type there are 10 possible type sequences: xxxyy **xxyxy xxyyx xyxxy xyxyx xyyxx yxxxy yxxyx yxyxx yyxxx** of which 20% of the sequences have the minimum number of changes of type.

Figure 2



The measured percentage of **struct** definitions having a minimal type change sequence (x-axis) and the percentage that are expected to occur if fields were ordered randomly. A cross indicates a measured percentage and the random percentage for a definition containing that particular number of types. If field ordering were random the crosses would be expected to cluster along the solid diagonal line (bullets indicate 1 standard deviation, dashed line 3 standard deviations). Data for all definitions containing between four and seven (inclusive) fields.

Table 1 lists possible type sequences for aggregates containing 4, 5 and 6 fields, their number of occurrence in C **struct** definitions, the expected percentage having the minimal number of changes of type and the actual percentage having minimal changes of type. The enumeration of field type sequences was calculated using a purpose-written program [3].

All pointer types were treated as being the same type (i.e., the **pointer** type), all arrays were treated as having the same type (i.e., the **array** type) and all struct types were treated as having the same type (i.e., the **struct** type; it did not matter whether these types occurred via the use of a **typedef** on an inline definition).

The number of **struct** definitions extracted from the source used for this analysis was sufficient to provide enough data to analyse definitions containing seven or fewer fields. Figure 1 shows a rapid, power-law like, decline in the number of C **struct** definitions as the number of fields in a definition increases, so significantly more source code would be needed to analyse definitions containing eight or more fields.

In all cases the number of definitions containing minimal field type change sequences was greater than would have occurred had the selection been random, Figure 2 shows the percentage is greater than three standard deviations (calculated as *sqrt(p(1-p)n)* where *p* is the probability of a minimal field sequence occurring and *n* is the number of instances, which has been normalised to 100).

Future research might analyse definitions where the number of type changes was only one, or more, greater than the minimum.

Table 1

Field type patterns for **struct** definitions containing various numbers of fields, the number of measured occurrences in source code and the measured percentage having a minimum number of type changes, and the percentage of type changes expected if field ordering was random. When all fields have the same type, or where each field has a different type, every sequence contains the same number of type changes and were not counted.

Fields	Field type pattern	Source occurrences	% minimum type change	% random ordering
4	112	239	77.4	50.0
4	13	185	78.9	50.0
4	22	98	62.2	33.3
5	1112	57	87.7	40.0
5	113	94	64.9	30.0
5	122	86	57.0	20.0
5	14	121	76.0	40.0
5	23	94	61.7	20.0
6	11112	19	73.7	33.3
6	1113	23	60.9	20.0
6	1122	28	35.7	13.3
6	114	53	64.2	20.0
6	123	72	55.6	10.0
6	15	51	66.7	33.3
6	222	9	100.0	6.7
6	24	60	51.7	13.3
6	33	21	57.1	10.0

Possible reasons for this grouping of fields by type include:

- Fields are grouped together semantically and such related fields are likely to hold similar kinds of information and this is represented using the same type.
- Developers are attempting to minimise unused storage space. Some processors require that types larger than a byte be assigned an address that is a multiple of some power of two. For instance, an **int** may have to be assigned an address that is a multiple of 2 or 4. Grouping fields having the same type is a simple strategy to ensure that no unused padding appears between these fields.
- Developers treat the type of a field as a categorization attribute and give it significant weight when deciding the relative ordering of fields within a structure type.

Same information, same name and type?

Developers are encouraged to give meaningful names to identifiers. If fields in different definitions have the same name (and contain a reasonable number of characters) it might be supposed that the information they hold is closely related semantically and these fields might be expected to share the same type.

This supposition relies on different developers representing the same information using the same name and the same type (the source code used for this analysis was written by a large number of developers). An earlier ACCU experiment [4] that asked subjects to assign a meaning to different identifier names found a wide range of meanings assigned to each identifier. The analysis of the experimental results, in part two of this article, will provide information on the extent to which developers choose the same name and type to denote the same application information.

Measurements of the translated C source show that within the measured 6,244 `struct` definitions and 47,554 fields, 21,805 (45.9%) fields had names that were not shared by other fields, while 6,415 names were shared by two or more of the other 25,749 fields (average 4.0 fields per name).

Table 2

Table 2 shows the number of times a given name occurs as a field and the number of different names occurring that number of times (e.g., if `blah_blah` is used three times as the name of a field it would add one to the count appearing in the second column of the 3's row). Only field names containing 4 or more characters were analysed.

Times name used	Number of different names	Percentage of all shared names
2	3239	50.54
3	1108	17.29
4	622	9.71
5	310	4.84
6	290	4.52
7	162	2.53

Is a field that shares its name with another field more likely to have a certain type? With one possible exception Table 3 suggests that the type of a field does not change the probability of its name appearing in another `struct` definition. A possible exception is pointer types which appear to be slightly more likely to occur, than the average, as the type of a field whose name appears in another `struct` definition (your author is uncertain how to appropriately calculate a standard deviation for values appearing in the two columns and so cannot say anything about statistical significance).

Table 3

Occurrences of structure type changes as a percentage of all field types (second column) and as a percentage of all fields whose names also appear in other `struct` definitions.

Type	% all occurrences	% duplicate occurrences
int	16.1	16.8
pointer-to	14.1	18.3
unsigned char	11.6	11.1
unsigned int	10.3	8.7
array of	9.4	10.9
unsigned short	7.6	7.5
struct	7.2	7.3

A study by Anquetil and Lethbridge [5] analysed 2 million lines of Pascal (see Table 4) and found that members that shared the same name were found to be much more likely to share the same type than members having different names.

Table 4

Number of matches found when comparing between pairs of members contained in different Pascal records. Adapted from Anquetil and Lethbridge.

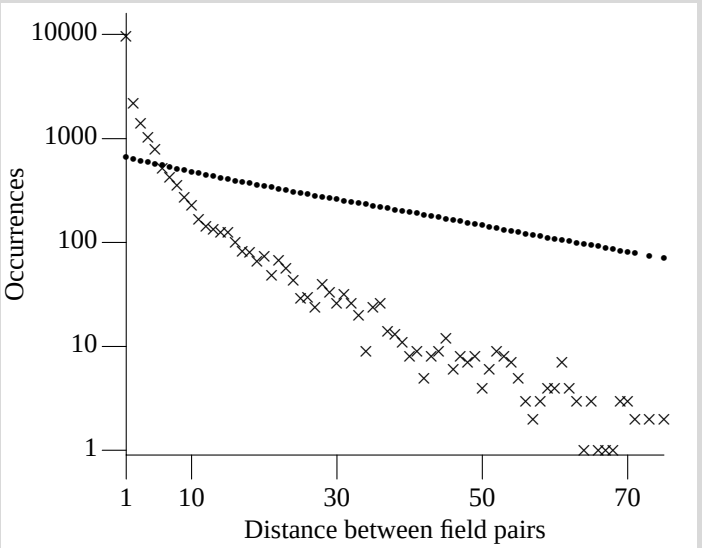
	Member Types the Same	Member Types Different	Total
Member names the same	7,709 (33.7%)	15,174 (66.3%)	22,883
Member names different	158,828 (0.2%)	66,652,062 (99.8%)	66,710,890

Refactoring and the desire for backwards compatibility can result in fields with the same name having the same type, i.e., fields moved into a new definition are retained in the original definition for backwards compatibility (perhaps because it is contained in a header included by lots of third party programs).

Fields sharing a subsequence of characters

Developers sometimes use several words or abbreviations to create an identifier having what they consider to be an appropriate semantic interpretation. If two or more fields within an aggregate definition share a semantic association it is possible that one or more subcomponents of their names will share a sequence of characters, e.g., `farm_house` and `farm_animal` both contain `farm`. The article on the 2007 ACCU experiment [6] describes an algorithm for extracting the likely intended subcomponents of an identifier and this was used for the analysis described here.

Figure 3



The number of individual field pairs that share one or more subcomponents plotted against their relative distance from each other; based on the translated C source (crosses) and based on random ordering of field pairs (bullets). Only definitions containing five or more fields were analysed and only character sequences containing three or more characters were counted. A field pair was only counted once, i.e., field pairs sharing more than one subcomponent were not counted multiple times.

A pair of fields within the same aggregate definition that share one or more is subcomponents is called a *field pair* in this article.

If developers group together semantically related fields within a definition, then field pairs will be closer to each other than if fields were ordered randomly. This grouping expectation relies on there being a sufficiently large number of different subcomponent character sequences that developers are not likely to assign different semantic interpretations to the same sequence.

Figure 3 shows the number of individual field pairs (crosses) that share one or more subcomponents plotted against their distance from each other (adjacent fields have distance 1, if one field separates them they have distance 2 and so on; see Table 5). The bullets show the behaviour expected if the fields in a field pair are ordered randomly.

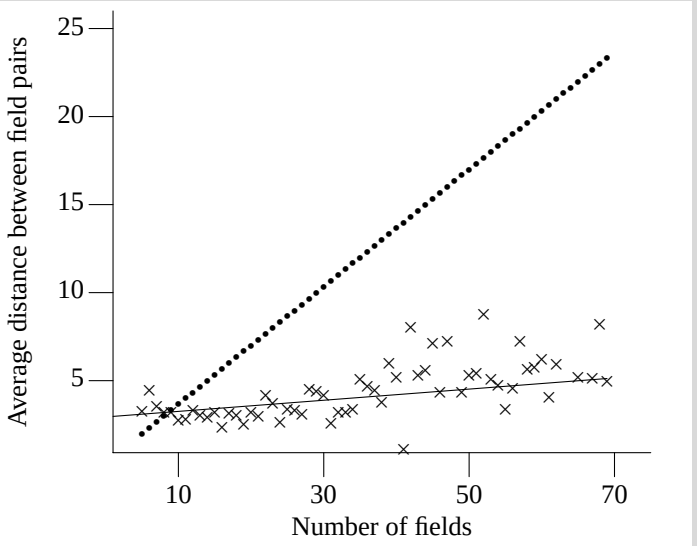
Figure 4 shows the average distance of all field pairs (crosses) occurring within a definition having a given number of fields. The bullets show the average distance expected if the fields in a field pairs are ordered randomly.

The average distance slowly increases as the number of fields in a definition increases. Reasons for this increase, rather than staying essentially the same, include:

- Definitions containing lots of fields are more likely to contain accidental field pairs, in the sense that they share a subsequence that was not explicitly intended to be shared, than definitions containing few fields.

- The greater the number of fields the greater the probability that many semantically related fields will be clustered together, creating field pairs over a greater distance.
- Fields are added and deleted from definitions as software evolves [7]. In some cases developers have to add new fields at the end of a definition so as not to change the storage layout of the existing fields. In this case a newly added field may create a field pair but not be placed much further away from its corresponding member(s) than would be the case if it had been added when the definition was first created.

Figure 4



The average distance between field pairs for definitions containing a given number of fields; based on the translated C source (crosses) and based on random ordering of field pairs (bullets). The straight line is a least squares fit of the average distance measurements. Only definitions containing five or more fields were analysed and only character sequences containing three or more characters were counted. A field pair was only counted once, i.e., field pairs sharing more than one subcomponent were not counted multiple times.

If field pairs are clustered together it is to be expected that the average field pair distance will be less than the random field distance for all sizes of definitions. Figure 4 shows that this is not the case when the definition contains between five and eight fields: 5 fields, expect less than 2.0 but find 3.25; 6 fields 2.33 vs. 4.46; 7 fields 2.67 vs. 3.54; 8 fields 3.00 vs 3.18. Possible reasons for this behaviour include: for smaller definitions developers feel that the semantic association between fields is more obvious than larger definitions and there is less need to explicitly call it out via shared subcomponent names; in a set of fields having a tight semantic association any shared name is likely to be part of the aggregate type name and repeating this sequence in field names would be overkill.

Average distance between random pairs of fields

The average distance between a randomly chosen pair of fields in a definition containing *N* fields is *(N+1)/3*.

Table 5

Number of possible occurrences (column values) of each distance (top row) between every possible combination of field pairs in definitions containing a given number of fields (left column). For instance, a `struct` containing five fields has four field pairs having distance 1 from each other, three distance 2, two distance 2, and one field pair having distance 4.

	Distance					
Number of fields	**1**	**2**	**3**	**4**	**5**	**6**
4	3	2	1			
5	4	3	2	1		
6	5	4	3	2	1	
7	6	5	4	3	2	1

Proof: the average distance can be obtained by summing the distances between all possible field pairs and dividing this value by the number of possible different pairs.

Table 5 shows the pattern that occurs as the number of fields in a definition increases.

In the case of a definition containing five fields the sum of the distances of all field pairs is: (4*1 + 3*2 + 2*3 + 1*4) and the number of different pairs is: (4+3+2+1). Dividing these two values gives the average distance between two randomly chosen fields, e.g., 2.

Summing the distance over every field pair for a definition containing 3, 4, 5, 6, 7, 8, ... fields gives the sequence: 1, 4, 10, 20, 35, 56, ... This is sequence A000292 in the On-Line Encyclopedia of Integer Sequences (www.research.att.com/~njas/sequences) and is given by the formula *n*(n+1)*(n+2)/6* (where *n=N-1*, i.e., the number of fields minus 1).

Summing the number of different field pairs for definitions containing increasing numbers of fields gives the sequence: 1, 3, 6, 10, 15, 21, 28, ... This is sequence A000217 and is given by the formula *n*(n+1)/2*.

Dividing these two formula and simplifying yields *(N+1)/3*.

Influence of a specification

Many software development projects start with some form of specification containing information about the application. This specification may be sufficiently detailed that it is possible to create a one-to-one mapping from the information it contains to an aggregate definition. Of necessity any information present in a specification, that may be encoded in the fields of a definition, appears in some order. Definitions are created by adding one field at a time and if the developer works systematically through a specification it is to be expected that field ordering used will have that order. Following the order of information in a specification has the following advantages:

- simplifies the process of verifying that all items listed in the specification are covered by a field,
- requires less cognitive effort to make use of an existing ordering than to create a different one,
- people influenced by what appears to be an existing way of doing things.

To what extent does the order of fields in `struct` definitions in existing code follow the order of corresponding information within a specification. Source of data that can be used to answer this question are hard to come by. National and international standards sometimes go into the level of detail required and in some cases there are implementations available whose source code can be viewed. For instance, POSIX [8][9] provides a specification of the information that a conforming implementation is required to support in various `struct` definitions. POSIX is supported by a wide variety of vendors who need to make available to their customers the source code, in header files, of the specified definitions. [10]

There are a number of complicating factors involved in using POSIX for this analysis, including:

- Implementations of much of the functionality specified in POSIX were available before work started on this standard. The authors of the standard may have based some of the orderings given in the specification on one or more of these existing implementations.
- One of the original vendors (i.e., AT&T) licensed various versions of its implementation to other vendors. It is possible that this resulted in some of the contents of some headers sharing more in common than they might have if implemented from scratch.

Header files from five different platforms supporting the functionality specified by POSIX were analysed (value in parentheses is the latest year any of the headers, for that implementation, could have been modified). The platforms were: RedHat 5.0 Linux (1996), Solaris 2.5 (1996), HP/UX 10 (1996), RS/6000 running AIX 3.2 (1992), Suse 10.3 Linux (2008). Two versions of the POSIX Standard were used for the analysis, the first [8]

published in 1990 and the latest [9] published in October 2008 (currently a final draft being voted as the next revision of POSIX).

Most of the **`struct`**s are required to support a relatively small number of fields (the number in parentheses): **`sigaction`** (3), **`uname`** (5), **`tms`** (4), **`stat`** (10), **`flock`** (5), **`termios`** (5) and **`passwd`** (5).

- With the exception of Solaris, which had one field out of order, all implementations of the **`sigaction`**, **`uname`** and **`tms`** structure types had the same field ordering as that given in the POSIX standard.
- In all implementations the **`flock`** (one additional field in Solaris, two additional fields in AIX) and **`termios`** (the same three additional fields in RedHat and Suse) structure types had the same field ordering as that given in the POSIX standard.
- Within the **`stat`** structure type defined in the header `sys/stat.h` the relative positions of the fields **`st_mode`** and **`st_dev`** were reversed between the first and latest edition of the POSIX Standard. Some vendors follow one ordering while the remaining vendors follow the other. All vendors have defined more than the ten fields listed in the first edition and intersperse the additional fields at a variety of different locations.
- Within the **`passwd`** structure type defined in the header `pwd.h` all of the implementations included fields not specified in either version of the POSIX standard, with the additional field shared across implementations always occurring in the same relative location.
- The **`sigevent`** structure type was not specified in the first edition of the POSIX standard, but is in later versions. It is defined in Solaris and HP/UX (in both cases with one field missing) and Suse (with two fields appearing in a different order).

Other definitions

It is likely that aggregate definitions will not be the only kind of definitions created by subjects taking part in the experiment. Names may be given to scalar types through the use of **`typedef`** (or some other mechanism appropriate to the language used by the subject) and names may be defined to represent particular kinds of entities (e.g., using macros or enumeration constants).

These non-structure definitions are not of interest to the questions investigated in this study, but they may be of interest for other questions concerning developer decision making. For instance, whether to use a macro (e.g., a **`#define`**; one study [11] was able to automatically map some sequences of identifiers defined as object-like macros to members of the same, tool created, enumerated type).

Discussion

The first part of this study set out to investigate the extent to which various patterns of **`struct`** field usage occur within existing source code.

- Fields that have the same type were found to occur in sequence with a probability that is significantly better than chance. There is no data to support any conclusions for why this might be so (e.g., developers actively try to achieve such a clustering or that clustering is driven by developer perceived semantic similarity between the information contained in fields and semantically similar information tends to have the same type).
- Fields in different definitions are sometimes given the same name (see Figure 2). There is no data to support any conclusions (e.g., developers use the same name to represent semantically similar information or the probability that different developers happen to use the same names). Fields having a particular type were not more likely to share the same name (see Figure 3). Refactoring and a desire for backwards compatibility can generate fields of the same name having the same type.
- Pairs of fields, in the same definition, whose names share a common subcomponent are much closer to each other than would be expected from a random field ordering. One explanation is that fields sharing a semantic association are grouped together and share one or more sequences of characters in their name.
- In a very small sample there was very good agreement between the ordering of information in a specification and the fields in structure definitions used by various implementations.

Threats to validity

The C source measured for this study has been actively worked on for many years and during that time its **`struct`** definitions are likely to have evolved. A study [7] of the release history of a number of large C programs, over 3-4 years (and a total of 43 updated releases), found that in 79% of releases one or more existing structure or union types had one or more fields added to them, while structure or union types had one or more fields deleted in 51% of releases and had one or more of their field names changed in 37% of releases. One or more existing fields had their types changed in 35% of releases [12]. The only source code measured was written in C. To what extent is it possible to claim that the findings apply to code written in other languages? Measurements of classes [13] in large Java programs have found that the number of members follows the same pattern as that measured in C (see Figure 1). While there are no obvious reasons why the patterns found in C should not also occur in other languages, there is no reason why they should. Measurements of source written in other languages would put this issue to rest.

Further reading

A readable collection of papers on how people make use of categories to solve problems quickly without a lot of effort: *Simple Heuristics That Make Us Smart* by Gerd Gigerenzer, Peter M. Todd and The ABC Research Group, published by Oxford University Press, ISBN 0-19 154381-7.

A readable upper graduate level book dealing with how people create and use categories *Classification and Cognition* by W. K. Estes, published by Oxford University Press, ISBN 0-19-510974-0.

Acknowledgements

The author wishes to thank everybody who volunteered their time to take part in the experiments and ACCU for making a slot available, in which to run the experiment, at both conferences.

Thanks to Faye Williams, Dawn Lawrie and David Binkley for commenting on an earlier draft and to Dawn for writing a program to evaluate field type sequences. ■

References and notes

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

[2] W. T. Maddox and C. J. Bohil. *Costs and benefits in perceptual categorization. Memory & Cognition*, 28:597–615, 2000.

[3] This program was written by Dawn Lawrie. Your author is keen to hear from anybody who knows of an algorithm, other than enumeration, for calculating these values. The number of different type combinations for a definition containing *N* fields is the number of integer partitions of *N*. The probability of encountering a field order having the minimum number of changes of type is obtained by dividing the number of minimum changes of type (this is *m!*, where *m* is the number of types in the partition) by the number of possible combinations of each type in the partition, for each of the partitions possible for a given definition.

[4] D. M. Jones. 'I mean something to somebody'. *C Vu*, 15(6):17–19, Dec. 2003.

[5] N. Anquetil and T. Lethbridge. 'Assessing the relevance of identifier names in a legacy software system'. In *Proceedings of CASCON'98*, pages 213–222, 1998.

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

This 'Software' Stuff, Part 2

Pete Goodliffe continues to unravel the meaning of (a programmer's) life.

In the first part of this mini-series we discovered money spinning ideas for people with strange eating habits, and started to take a slightly philosophical look at what the act of programming involves. We're trying to understand exactly what this 'Software Stuff' is so we know how to build it better – how to write the right thing in the right way. We took inspiration from the mythical ideal programmer, saw that programming is an art.

In this part we'll see how programming is both a science and a sport. Remember that as we go along I will be posing a series of personal questions. Consider each question and see whether it applies to you.

Software is... a science

Well, we talk about Computer Science. So there must be something vaguely scientific going on, mustn't there? Although it's probably fair to say that in day-to-day development organisations there is much less science and more plumbing involved.

The archetypal mad scientist is, of course, Mr Albert Einstein. He was great – about the most quotable scientist there has ever been (which helps magazine authors considerably). And he had a great sense of humour. He said this: Any intelligent fool can make things bigger, more complex, and more violent. It takes a touch of genius – and a lot of courage – to move in the opposite direction.

That's a really profound quote; inappropriate complexity is a real killer in most software projects.

Harking back to Part 1, Einstein was a bit of an artist too. He appreciated elegance and beauty in his theories, and aimed to reduce things to a coherent whole. He said: I am enough of an artist to draw freely upon my imagination. Imagination is more important than knowledge. Knowledge is limited. Imagination encircles the world.

See, I told you he was quotable.

So if software is like a science, was does that mean? It is (or should be)...

- **Rigourous** We look for bug-free code that works, all the time, every time. It must work with all sets of valid input, and respond appropriately to invalid input. Good software must be accurate, proven, measured, tested and verified. How do we achieve this? Good testing is key. We look for unit tests, integration tests, system tests. Preferably automated to remove the risk of human error. We also look for experiential testing, too.
- **Systematic** Software development is not a hit-and-miss affair. You can't aim to create a well-structured large computer system by randomly accreting blobs of code until it appears to work. You need to plan, design, budget, and systematically construct. It is an intellectual, logical, rational, process; bringing order and understanding out of the chaos of the problem space and the design alternatives.
- **Insightful** Software development requires intense brain powers and astute analytical powers. This is especially important when tracking down tricky bugs. Like scientists, we form hypotheses, and apply something akin to scientific method (form hypothesis, work out experiments, run experiments, validate theory).

So, based on that, ask yourself:

> **Is my**... software always totally correct and completely accurate? How do I prove this? How can I make this explicit, now and in the future?

...and...

> **Do I**... strive to bring order out of chaos? Do I collapse complexity in my code until there are a few, small, unified parts?

...and...

> **Do I**... approach problems methodically and thoughtfully, or do I rush headlong into them in an unstructured way?

Software is... a sport

Not all metaphors are perfect, and I'll admit that we have to treat this one carefully. Here I'm not talking about 'loner' sports, like running. Before I alienate all the athletes in my readership, I go running every week, and

PETE GOODLIFFE

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

Developer categorization of data structure fields (continued)

[7] I. Neamtiu, J. S. Foster, and M. Hicks. 'Understanding source code evolution using abstract syntax tree matching'. In *Proceedings of the 2005 International Workshop on Mining Software Repositories*, pages 1–5, May 2005.

[8 ISO. ISO/IEC 9945-1:1990 'Information technology —Portable Operating System Interface' (POSIX). ISO, 1990.

[9] ISO. ISO/IEC FDIS 9945:2008 'Information technology —Portable Operating System Interface (POSIX®)'. ISO, 2008.

[10] Your author is keen to hear from anybody who knows of any other publicly available specifications and associated implementation headers.

[11] J. M. Gravley and A. Lakhotia. 'Identifying enumeration types modeled with symbolic constants'. In L. Wills, I. Baxter, and E. Chikofsky, editors, *Proceedings of the 3rd Working Conference on Reverse Engineering*, pages 227–238. IEEE Computer Society Press, Nov. 1996.

[12] I. Neamtiu. Detailed break-down of general data provided in [6] kindly supplied by first author. Jan. 2008.

[13] R. Wheeldon and S. Counsell. 'Power law distributions in class relationships'. In *Third International Workshop on Source Code Analysis and Manipulation (SCAM 2003)*, pages 45–54, Sept. 2003.

Figure 1

manage to solve a lot of programming problems that way! But that's not where I'm drawing parallels here. (As it happens, I also have some of my best programming ideas on the toilet, but that's an entirely different story).

Software development involves:

- **Teamwork** It requires many people, with different skills, working in harmony.
- **Discipline** Each team member must be in training, giving commitment to the team, and putting in hard work.
- **Rules** We're playing to (developing to) a set of rules, and a team culture. This is embodied by things like our development processes and procedures, as well as the rites and rituals of the software team and their tool workflows (consider how you collaborate around things like the source control system).

The teamwork analogy is clearest with a sport like soccer. You work in a team of closely functioning people, playing a game by a set of well-defined rules.

Have you seen a team of 7-year-olds playing soccer? There's one small guy left back standing in the goal mouth, and every other kid is running around the pitch maniacally chasing the ball. There's no passing. There's no communication. There's no awareness of the other team members. Just a pack of children converging on a small moving sphere.

Contrast that to a high-quality premier league team. They operate in a much more cohesive way. Good software teamwork involves a number of important skills, highlighted in Figure 1. These are just a few of the essential characteristics for good software development team work (I talk more about this in my book [1]). How good are you at each of these?

> **Do I**... have all of these skills? Do I work well in a team, or could I improve in some areas?

> **Do I**... want to improve as a programmer? Do I actually want to write the right thing in the right way?

Another key aspect of being a sportsman is discipline. It requires dedication, hard work, and a lot of training. You can't get good at soccer by sitting on a couch and watching soccer training videos. In fact, if you do it with a few beers and a tub of popcorn, you're likely to get worse at soccer! You have to actually do it, get out there on the pitch, with people, practise your skills, and then you'll improve. You must train – have someone tell you how to improve.

And the team must practise together, work out how individual people should interact, and practise those team moves. These are dictated in part by the rules of our game.

> **Am I**... still learning about software development? Do I learn from others, and am I perfecting my team skills?

> **Do I**... invest enough effort in my own development? Am I continually in training? Am I tired and muddy?

More anon

This is most definitely a whistle-stop tour of the world of software development, but I hope it helps to highlight some of the essential things we need to work at to become better software artists/scientists/sports(wo)men.

In the final part of the mini-series we'll think about more aspects of this incredible software 'stuff'. ■

References

[1] *Code Craft: The Practice of Writing Excellent Code*. Pete Goodliffe. No Starch Press, 2007. ISBN 1593271190.

UML: Getting the Balance Right

Aaron Ridout discusses the usage and aesthetics of UML.

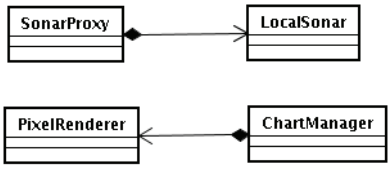
This is a quick article on a few areas of UML not normally covered in the main text books. Please direct any comments or discussion points via ACCU General (by the time you read this I should be back from paternity leave, holidays and TOIL!), and/or feel free to e-mail me directly.

Aesthetics

I have no scientific results, but I have a 'gut feeling' that says if a diagram looks balanced then this implies that the system being described is 'better' (by some measure of 'better'). Personally, I see this as an extension of the adage that if a class has an **`Open()`** method then it should also have its opposite: a **`Close()`** method. Likewise **`Do()`** and **`Undo()`**, but it is also based on the idea of symmetry in a diagram (about some axis or other).

Some layouts are better than others; the layout helps the user to understand the intent on a diagram. In English, we write from left to right, and thus have an intuitive understanding that a diagram has time running from left to right; so a class drawn to the left of another starts first, processes first, or manages the one to its right. I wonder if someone from other cultures would recognise this idiom in the same way? (See Figure 1, which compares left to right with right to left.)

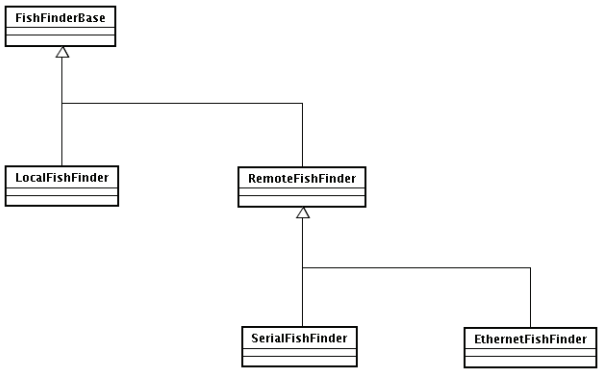
Figure 1



What do you think? Personally, the obvious interpretation is correct for the top pair, while the bottom pair requires slightly longer thought to intuit that the **`ChartManager`** is in charge. This gets harder as the class diagram contains more classes. I see this as akin to the signal-to-noise ratio: the fewer classes on a diagram the easier it is to understand. Unless, of course, you require an overview of the bigger picture, whence you (almost) want a one-of-all type diagram on paper sometimes as big as 10m square! I find that keeping to a 7 ± 2 rule [1] per diagram keeps the meaning and intent clearer, i.e limit each diagram to about 7 'interesting' nodes, with a total of no more than 20 interesting-plus-supporting nodes. I would also advocate the use of colour to show the main focal points, versus that which is supporting or peripheral to the intent of the diagram. Colour is also a good indicator of pattern membership, e.g. all members of a bridge pattern could appear as a particular shade of yellow.

I hope that it is obvious, but *don't cross the streams* (lines). The more lines that cross on a diagram, the harder it is to see the associations.

Figure 2

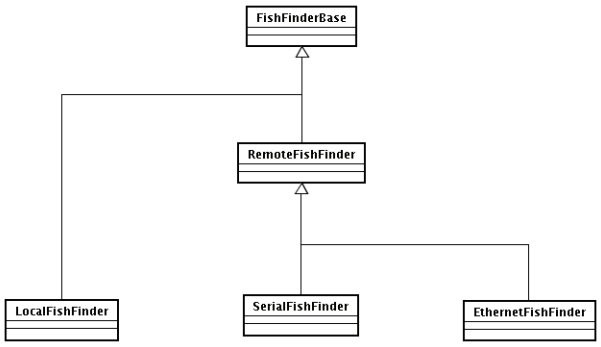


Cascade or mobile

Which of the two diagrams shown in Figure 2 (cascade inheritance) and Figure 3 (mobile inheritance) do you prefer? Does it make a difference? What if you wanted to make it clear that there are various layers of implementation details? Or were more interested in displaying that there are three leaf classes?

Personally, I think the best choice depends on what you are trying to show: layers or leaves.

Figure 3



Call-trees

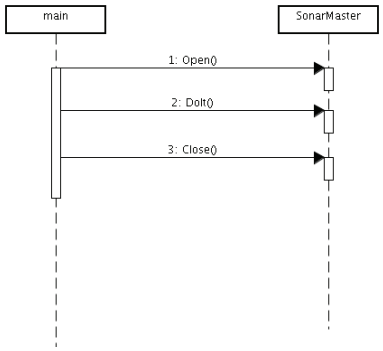
These need to be balanced also, but here I look for horizontal lines of symmetry. In Figure 4, a balanced call tree, there is a line of symmetry about the **`Doit()`** call, so all is well.

For some time I've agonised over Resource Acquisition Is Initialisaton (RAII): is it balanced? In the source code example below, one constructs a **`Locker`** object, but never actively destroys it; it just oozes out of scope, so the code looks unbalanced because there is no active 'unlock'.

```
void SonarMaster::DoIt() {
  Locker lock(mutex_);
  // do stuff while locked
} // lock goes out of scope here!
```

However, looking at the sequence diagram in Figure 5, the addition of the **`~Locker()`** line that the compiler inserts for you, makes the diagram look balanced to me – what a quandary!

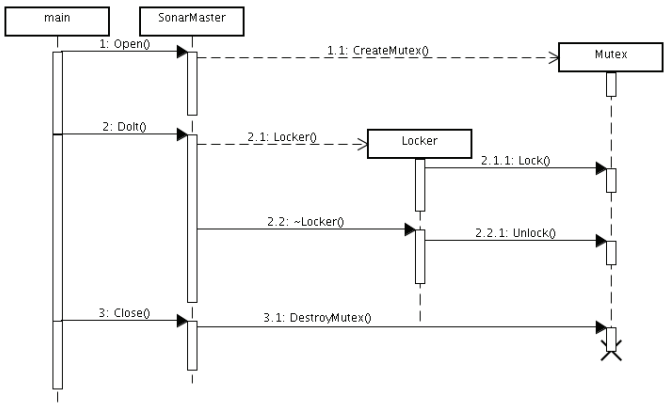
Figure 4



AARON RIDOUT

Aaron's love of colourful jumpers offsets his fascination with agile methods. Happily married with 17 children, an Acorn Atom and a suitcase full of ZX81 tapes, in another world he'd have been a children's TV presenter. Contact him at aaron.ridout@blackcat.co.uk.

Figure 5



Attributes and associations

At the end of the day, attributes and associations both map to the same things in your class: member variables. On a diagram I think you should always use associations unless there are too many to coherently display, in which case, omit the ones that are less interesting with regard to what you are trying to convey.

Alas, I don't know of any UML tool that let you display a member variable as an association on some diagrams and as an attribute on others. If you don't want the association on a particular diagram, don't put the target class on that diagram – information hiding!

Implementation language

An analysis diagram can be drawn in 'pure' UML with no regard to the language in which the system might be implemented. However, a design diagram that is closer to the actual code to be produced needs to be moulded to use the type system of the implementation language in mind. UML tools have a huge variation in how well they support any given implementation language: most offer some sort of one-way code generation, a few offer round-trip. Personally I've only ever successfully managed 'repeated one-way', i.e. the diagram is kept up to date, it generates the source code framework and one writes source code in between the special comments that the UML tool produces in the generated source. Few UML tools understand the twiddly bits like **`const`** or volatile type qualification (cv-qualifiers); you can often end up with several (un)related types like **`Item`**, **`Item *`**, **`Item const &`**, **`Item const *`**, **`Item volatile * const`**, etc. To us, the users of the UML tool, they are all the same type (**`Item`**) with various cv-qualifications; to the UML tool they are all separate types.

Language binding

Table 1 shows how I view the C++ language binding working as the strength of association decreases, i.e. at the top of the table the two classes are strongly coupled and at the bottom of the table they are minimally coupled. I've included some cutesy names to describe their relationship, but I'm open to suggestions of improvement for these!

If the upper bound (*n*), is known, then multiple composition could be implemented via C arrays, but I prefer **`std::vector`** as it provides better size and other container operations. Similarly, multiple aggregation could be a simple array of pointers to **`T`**, or it could utilise **`std::list`**, or some other container that fits your requirements. What you choose will depend on the code you want to produce, and some UML tools are more configurable than others with regard to the C++ code they will generate. For example, some UML tools let you use an association class and/or a stereotype to specify the container, such that you can state that a **`std::set<T>`** is used rather than a **`std::vector<T>`**. Whether this is important to you depends on how close your diagrams are to the actual code.

Semantic programming

A UML tool should easily support semantic programming, you just have to be careful to use packages correctly to scope the generated files, otherwise you'll end up with one file for each type and a huge compile time because the number of include files has rocketed.

Table 1

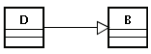
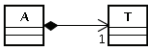
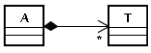
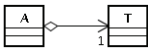
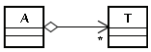
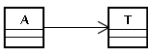
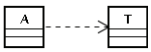
	Public Inheritance	**`d.h:`** **`#include "b.h"`** **`class D : public B`** **`{};`**	is-a
	Protected Inheritance	**`d.h:`** **`#include "b.h"`** **`class D : protected B`** **`{};`**	is-like-a
	Private Inheritance	**`d.h:`** **`#include "b.h"`** **`class D : private B`** **`{};`**	like-a
	Single Composition	**`a.h:`** **`#include "t.h"`** **`class A {`** **`T m_T;`** **`};`**	has-a
	Multiple Composition	**`a.h:`** **`#include "t.h"`** **`class A {`** **`std::vector<T> m_Ts;`** **`};`**	has-some
	Single Aggregation	**`a.h:`** **`class T;  //`** **`forward declaration`** **`class A {`** **`T * const m_pT;`** **`};`**	uses-a
	Multiple Aggregation	**`a.h:`** **`#include "t.h"`** **`class A {`** **`std::list<`** **`tr1::smart_ptr<`** **`t>> m_Ts;`** **`};`**	uses-some
	Association	**`a.h:`** **`#include "t.h"`** **`class A {`** **`friend t;`** **`};`**	refers-to
	Dependency	**`a.cpp:`** **`#include "t.h"`**	knows-of

Figure 6 shows that the Window class **`knows-of`** (includes) two enums, They can be in one file, separate files or in an include **`bucket`** (i.e. a bad smell that several places have: e.g. **`our_types.h`**).

Tools and patterns

One feature I would really like to find in a UML tool would be real and proper support for patterns. What I mean by this is the ability to select a pattern from a catalogue, say the bridge pattern, drop it onto my diagram and perhaps fill in a property page to, say, rename the classes and select a colour for the participants. I then expect that the tool will maintain the pre-

Figure 6

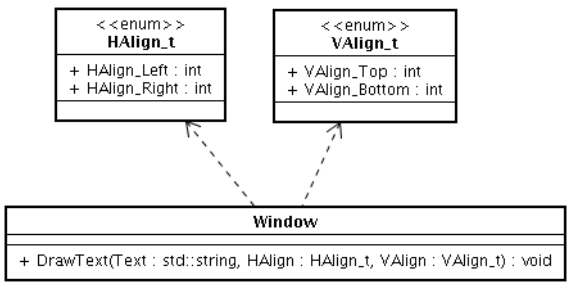


Table 2

Project	Language	Process	Staff	Cost(£K)	Effort (days)	SLOC	Classes	Effort/Class	Lines/Day	Cost/Class	Cost/Line
GByte/sec Data Logger	gnu C++	Waterfall	~12	£3,567	8394	100000	5000	1.679	12	£713	£36
MByte/sec Data Logger	Borland C++	Waterfall	4	£147	240	30000	200	1.199	125	£736	£5
Production Test system	Borland C++	Waterfall	7	£ 270	406	53000	411	0.987	131	£656	£5
Digital Tape Playback	MSVC++	Waterfall	9	£ 171	306	38000	336	0.910	125	£509	£4
Workflow (Iteration 2)	Java	XP	5	£ 156	232	14000	197	1.178	62	£790	£11
Workflow (Iteration 3)	Java	XP	7	£ 251	383	20000	304	1.260	54	£825	£12
Workflow (Iteration 4)	Java	XP	7	£ 309	477	29000	391	1.221	60	£791	£11

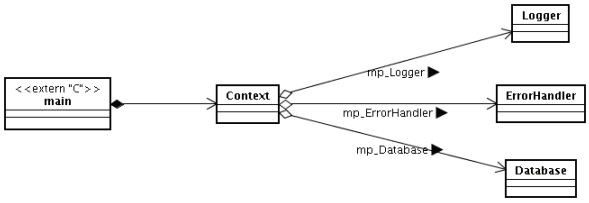
conditions, post-conditions and invariants of the pattern! I.e. if I add a `DoABC()` method to one side of the bridge, I expect the tool to automagically add the same method to the other side of the bridge, or at least offer to do so. This extends, obviously, to state charts and sequence diagrams and probably all other ~9 diagram types too!

Parameterise from Above (PFA)

On the subject of patterns, PARAMETERISE FROM ABOVE has been discussed a lot recently, so here's my £0.03…

Take the small example in Figure 7. If you have a smaller number of objects to create and pass around, hopefully they are created and owned by `main()`, or perhaps by lazy evaluation (but please not singletons, just don't do it!), so that here the Context object has pointers to its constituent parts (NB: forward references are used to minimise coupling, the relevant header files being included only in the cpp that needs to use that one module). This is so the Context object can be passed around or through layers without inflicting coupling side effects in all classes that the Context simply passes through. I've used this style successfully in a Java application of ~100k SLOC, but then in Java everything is a reference and everyone sees the object declarations.

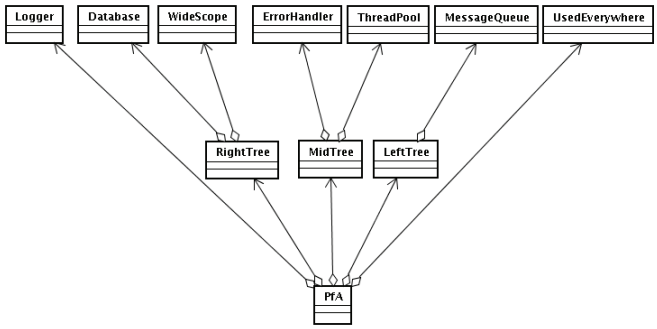
Figure 7



When the number of objects increases and their life-times are more complex, then you need to consider when and where they are created and how much coupling you would inflict when passing the parameters around. So I would consider having the parameters in different groups that can be split up as they are passed down a call tree, i.e the larger PFA is passed in at the top, and as you descend through the layers the smaller 'tree' groups are passed on – see Figure 8. This reduces coupling and means that the (smart) pointers have to be copied. Also, there could be duplicates: e.g. an `ErrorHander` in all three tree groups. Obviously my names are fictitious and as such this is a poor example (sorry), but this would be a hierarchy that would in some way reflect the 'tree' in which they were being passed around.

I have not used this in anger yet – it's very much a work in progress. In my model, `main()` creates all of the objects as it constructs the main PFA, which also means that the life-times of all the constituents are simply the same as main!

Figure 8



Tool gripes

UML tools are not perfect, as we all know. I would propose that 'Some tools do not use strong OO methods to design themselves'. What I mean by this sweeping generalisation is that I want to be able to do things like copy a method from one class (with all its parameters and settings) into another class, and I would also expect to be able to do deep or shallow copies, i.e. include any source code with it or not. This extends to class, attribute, association, etc.

I would also like to see the UML model of the tool shipped with the tool as an example UML model. This would allow users to write and integrate better scripts into the tools to do those little jobs the tool vendor hasn't yet implemented or that are not 'core' enough.

I remember tying to create a script for a well known tool that could change an attribute into an inheritance relationship. Simple, just change the line style from 'composition' to 'realise' – ha ha ha! How wrong can one be? Instead the process required taking a note of the details of the composition, deleting the old composition from the model (and trying to keep any orphaned classes on the diagrams they were on), then creating the new realise relationship, filling in any details that you had, and finally forcing the new line to appear on all diagrams that the participants were drawn upon!

In their own UML model there was absolutely no relationship between any of the flavours of 'inheritance' and any flavour of 'association', so you could change a member variable from association to composition, calling at all stations betwixt, but not into any form of inheritance; which to my mind is simply the next stop on the 'strength of association' scale (see Table 2).

While I'm on the subject of gripes, why do so few implementations of sequence diagrams generate any code? Or worse, why don't sequence diagrams get created when reverse engineering existing code? Doxygen can create them (I suppose all I need to do is output the XML from Doxygen, write a quick XSL script and output XMI…).

Which leads nicely on to XMI: why, oh why, don't UML tools import and export XMI in a compatible way? Is the standard too lax? Or do the tools not want you to import and export? Indeed, one of my evaluation criteria would be to round trip a complex model via XMI to see exactly how much information is lost.

Project estimates

Finally, I would like to finish with a simple exercise that may be of interest. Once you have a model you can (i.e. it may not be useful!) use it to run some statistics on the code you have generated. It is quite revealing to see how much a line of code theoretically costs. Table 2 is a summary of work I did some 15 years ago, but is obviously out of date; collect your own statistics! ■

References

[1] The Magical Number Seven, Plus or Minus Two. http://en.wikipedia.org/wiki/Hrair_limit

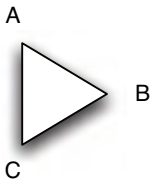
Acknowledgements

Many thanks to Neil Owens and Peter Cullinane for reviewing this article. Share and enjoy!

Containment During Subdivision

Thaddaeus Frogley tries to find the point.

This article describes an efficient and reliable method for determining which subdivision of a triangle contains a point.



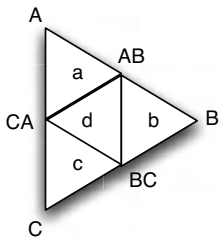
Given the triangle A–B–C, which can be subdivided into 4, so that:

a = A–AB–CA

b = AB–B–BC

c = AB–BC–C

d= AB–BC–CA



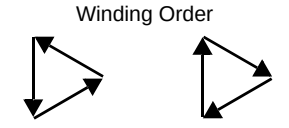
And given a point, P, inside ABC, which of a, b, c, d contains P? Note that although this article describes Sierpinski's subdivision of 2d triangles, the method described can be generalized to apply to other subdivision spaces.

Background

There are a number of ways of determining if a point lies within a triangle, such as using Barycentric Coordinates [1], an Angle Summation test, or checking which side of each edge the point lies. In order to better understand the final solution below, I will now explain how containment using edge-side tests works.

Given an edge, AB, which side of the edge the point P lies can be determined by checking the sign of (*AB.x* * *AP.y*) - (*AB.y* * *AP.x*), thus for a **`Line`** class that has a **`Start`** vector and a **`Finish`** vector, a **`Side`** function might look like this:

```
Scalar Side(const Vector& p)const
{
  Vector ab(this->mFinish - this->mStart);
  Vector ap(p - this->mStart);
  return (ab.GetX()*ap.GetY())-
     (ab.GetY()*ap.GetX());
}
```



A point lies inside a single triangle if it is on the right side of all of its edges. Note, 'right' becomes left if the winding order is anti-clockwise. Thus a containment test for a triangle, using the side-of-line technique, might look like this:

```
bool Triangle::Contains(const Vector& p) const
{
  return (mAB.Side( point ) >= 0) &&
         (mBC.Side( point ) >= 0) &&
         (mCA.Side( point ) >= 0);
}
```

The obvious solution to the subdivision problem, testing each subdivision to see if the point lies within it, does not work.

THADDAEUS FROGLEY

Thaddaeus Frogley is a Senior Programmer at Climax, a games developer based in Portsmouth, England. Contact him at codemonkey.uk@gmail.com

Why 4x point inside triangle doesn't work

ISO/IEC 14882 3.9.1/8 defines the precision of float, double, and long double relative to each other. No absolute guarantee of numerical accuracy is specified. In practice it is likely that the floating point arithmetic types will be IEEE 754-2008 floats, of 32, 64, and 128 bits respectively, and the results of operations will be rounded to fit in that number of bits.

In addition to imprecision from rounding, the IEE standard only recommends that operations are reproducible (that is to say, the same input is not guaranteed to produce the same output). This means that when dealing with the results of operations on arbitrary floating point values, you cannot depend on **`(a*b==a*b)`**, and you certainly can't expect **`a*b`** to be binary equal to **`b*a`**.

What this means for us in practice is a containment test for a point very close to a shared edge could be on the inside of both, or on the outside both of the triangles that it borders, causing the point to 'fall through the crack' and thus the test to fail.

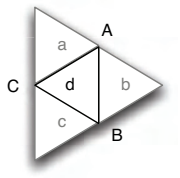
For further reading on the floating point issue, see David Goldberg's paper 'What Every Computer Scientist Should Know About Floating-Point Arithmetic' [2].

It is often suggested that bugs stemming from rounding errors could be avoided by using a higher precision type, be it switching from float to double, or using an arbitrary-precision arithmetic library, but this is both unnecessary and ineffective, since it only delays the inevitable.

Solution

The solution to the problem is to consider only the edges that make up subdivision **d**:

- if *p* falls to the left of AB, it is inside **b**
- if *p* falls to the left of BC, it is inside **c**
- if *p* falls to the left of CA, it is inside **a**
- otherwise, it must therefore be inside **d**.



Example code:

```
Triangle* subD = &tri->GetSubDivision( tD );
if (subD->GetEdge2dXY( tAB ).Side( p ) < 0)
   tri = &tri->GetSubDivision( tB );
else if (subD->GetEdge2dXY( tBC ).Side( p ) < 0)
   tri = &tri->GetSubDivision( tC );
else if (subD->GetEdge2dXY( tCA ).Side( p ) < 0)
   tri = &tri->GetSubDivision( tA );
else tri = subD;
```

Conclusion

By using a process of elimination, we end up with a much more satisfactory solution than the obvious series of tests seeking a positive result. Rather than asking where the point is, we have asked where it is not. ■

Notes

[1] http://en.wikipedia.org/wiki/Barycentric_coordinates_(mathematics)#Determining_if_a_point_is_inside_a_triangle

[2] http://www.validlab.com/goldberg/paper.pdf

Boiler Plating Database Resource Cleanup

Paul Grenyer cleans up after his code.

I've been using Java for nearly twelve months now and I am finding that I like it. There are only two things that I have discovered so far that make make me wonder what the creators of Java were thinking: exception handling and layout managers. I'll cover layout managers in a later article.

Java is a garbage collected language. Which means that, most of the time, you don't have to worry how the memory previously used by dead objects is cleaned up. To me garbage collection has always felt a bit like a knee jerk reaction for people who can't use smart pointers properly in C++, and for C programmers who must pay very close attention to the points at which they release memory allocated to the heap. If garbage collection is meant to help you clean up memory, why hasn't something been developed to help objects release resources? Java has finalizers, but as Joshua Bloch points out in item 7 of *Effective Java* [1] finalizers cannot be relied upon (see sidebar 1). Proper cleanup of resources is left to the developer whose only real friend is `finally`.

In languages like C# the `IDisposable` [2] interface can be used to aid cleanup. It has a single method called `Dispose`. All of an object's cleanup code should be placed in this method, and calling the method manually, or via `using`, ensures resources are released. The drawback is that `IDisposable` requires the client code to to use it. There is nothing like this available in Java, but I think I may have an even better solution which takes away the dependence on a class's user to do any cleanup.

Tony Barrett-Powell in 'Handling Exceptions In Finally' [3] and Alan Griffiths in 'Exceptional Java' [4] and 'More Exceptional Java' [5] both discuss methods of dealing with database related cleanup and exceptions, but their solutions are still verbose and can be boiler plated. These articles form the basis of the boiler plate I present here.

In this article I'll look briefly at the vast amount of exception handling code that must be written to cleanly and efficiently close result set, statement and connection objects, without relying on finalizers when accessing a database, and then more in depth at one possible method of boiler plating it.

The problem

The problem is simple. Cleaning up after querying a database in Java is unnecessarily verbose and complex. Plain and simple. I'll start with an example that demonstrates the problem. The system I'm working on currently uses a number of web services. We have a set of web services on the production box, another on the development box and another on our local machines. The system asks the relevant database for the location of the web service based on the services' name (Listing 1).

This is a lot of code to get one string out of a database and most of it must be repeated every time a database is accessed. Most of it is error handling and resource management. In fact I've over simplified it. For a discussion of how error handling and resource management should really be handled see 'Handling Exceptions in Finally' mentioned above.

The Sun Java Documentation [6] states the following for the `Connection` interface's `close` method:

> Releases this `Connection` object's database and JDBC resources immediately instead of waiting for them to be automatically released.
>
> Note: A `Connection` object is automatically closed when it is garbage collected. Certain fatal errors also close a `Connection` object.

It also states the following for the `Statement` interface's `close` method:

> Releases this `Statement` object's database and JDBC resources immediately instead of waiting for this to happen when it is automatically closed. It is generally good practice to release resources as soon as you are finished with them to avoid tying up database resources.
>
> Note: A `Statement` object is automatically closed when it is garbage collected. When a `Statement` object is closed, its current `ResultSet` object, if one exists, is also closed.

And the following for the `ResultSet` interface's `close` method:

> Releases this `ResultSet` object's database and JDBC resources immediately instead of waiting for this to happen when it is automatically closed.

'Effective Java'

Item 7: - Avoid Finalizers

This item could have been written to put me straight. I'm the C++ programmer it speaks about and I was desperate for finalizers to be Java's destructor. They aren't. In fact, unlike C++'s destructors, they are unpredictable, often dangerous and generally unnecessary. Ok, so if you're not careful with exceptions, destructors in C++ can be dangerous too. Java finalizers are worse.

The item explains that you should never do anything time critical io finalizers as the JVM is 'tardy' at running them. I did some experiments closing database connections in finalizers and couldn't generate any evidence that they were called at all. If and when finalizers are called is JVM implementation specific. So cross platform programming using finalizers is unpredictable at best, disastrous at worst.

The item also explains that uncaught exceptions thrown in finalizers are ignored, but the finalization of the object is terminated leaving it in an unknown and potentially corrupt state, which can result in arbitrary, nondeterministic behavior. Using finalizers also increases the time to terminate an object by a whopping 430 times according to an (unexplained) test example run by Bloch.

The item describes the alternative to using finalizers as providing an explicit termination method that can, typically, be called from a finally block. This is what I do in a lot of cases and what the Java database objects provide.

Finalizers could be used as a safety net for forgotten terminate method invocations on the basis that it's better to release resources late than never, but of course there's no guarantee it'll get released at all as there is no guarantee that a finalizer will ever be called...

Item 65: - Don't Ignore Exceptions

The point that this item is trying to get across is that exceptions are trying to tell you that something bad has happened and should not be ignored. It's far too easy to write something like:

```
try
{
  …
}
catch(Exception e)
{
}
```

The item points out that 'an empty catch block defeats the purpose of exceptions, which is to force you to handle exceptional conditions' and 'At the very least, the catch block should have a comment explaining why it is appropriate to ignore exceptions.'

The item gives the example of closing a `FileInputStream` as a situation where it might be appropriate to ignore an exception, with a comment in the catch block or a message (written to a log of course). The state of the file has not been changed and there is nothing to roll back. This of course assumes that the file resource has been successfully released.

The advice in this item applies equally to checked and non-checked exceptions.

PAUL GRENYER

An active ACCU member since 2000, Paul is the founder of the Mentored Developers. Having worked in industries as diverse as direct mail, mobile phones and finance, Paul now works for a small company in Norwich writing Java. He can be contacted at paul.grenyer@gmail.com

Listing 1

```
try
{
   Class.forName(driver);
   Connection con = DriverManager.getConnection(
      connectionString, username, password );
   try
   {
     PreparedStatement ps = con.prepareStatement(
        "select url from services where name =
        'Instruments'");
     try
     {
       ResultSet rs = ps.executeQuery();
       if(rs.next())
       {
         System.out.println(rs.getString("url"));
       }
       try
       {
         rs.close();
       }
       catch(SQLException e)
       { // Report Error }
     }
     finally
     {
       try
       {
         ps.close();
       }
       catch(SQLException e)
       { // Report Error }
     }
   }
   finally
   {
     try
     {
       con.close();
     }
     catch(SQLException e)
     { // Report Error }
   }
}
catch(Exception e)
{ // Report Error }
```

Note: A **ResultSet** object is automatically closed by the **Statement** object that generated it when that **Statement** object is closed, re-executed, or is used to retrieve the next result from a sequence of multiple results. A **ResultSet** object is also automatically closed when it is garbage collected.

This can all be interpreted in a number of ways. The first is that everything gets closed when it is garbage collected, so there is no need to explicitly close anything. This relies on the appropriate finalizers getting called but, as Bloch tells us, Java provides no guarantee that a finalizer will ever be called, even when an object is garbage collected.

Another method is to explicitly close **Statement** and **Connection** objects as **Statement** objects clean up their associated **ResultSet** objects. Drawbacks include any error caused by closing the **ResultSet** is potentially ignored and the resource is not released as soon as it could be.

Yet another method is to explicitly close everything. This is the most code, but releases resources and handles any error as soon as a resource is finished with, making it the most efficient way of using resources.

I favour the third and final method. It is more code, but boiler plating will mean most of it only needs to be written once. Everything will be cleaned up as soon as possible, all errors can be trapped and reported, and nothing is left to chance.

Listing 2

```
public class ConnectionException extends
   ResourceHandlerException
{
  public ConnectionException(
     final String message,
     final Throwable throwable)
  {
    super(message, throwable);
  }
}
```

The **close** methods for **Connection**, **Statement** and **ResultSet** objects can all throw if there is an exceptional circumstance. In Item 65 of 'Effective Java' (see sidebar), Bloch explains that exceptions should not be ignored. He also describes certain circumstances where it *might* be okay to ignore or log these types of exceptions. The **Connection**, **Statement** and **ResultSet** close methods could be considered one of these situations.

A solution

In 'Another Tale of Two Patterns' [7] Kevlin Henney describes the FINALLY FOR EACH RELEASE and EXECUTE AROUND METHOD (EAM) patterns. Both are applicable to the problem. Here I will look at the FINALLY FOR EACH RELEASE solution and in Part II a solution using EXECUTE AROUND METHOD.

In its basic form the FINALLY AFTER EACH RELEASE pattern looks like this:

```
resource.acquire();
try
{
  resource.use();
}
finally
{
  resource.release();
}
```

which describes accurately how database resources should be created, used and cleaned up.

Connection policies

Let's start by looking at database connection management. There are a number of ways of creating and using a database connection. Probably the two most common are:

- Creating it with a driver (e.g. JDBC) and a connection string.
- Using an existing connection created and cleaned up somewhere else.

Sensible boiler plate code should support both of these methods and allow custom creation of database connections, which makes using a policy ideal:

```
public interface ConnectionPolicy
{
  abstract Connection connect()
     throws ConnectionException;
  abstract void disconnect(final Connection con)
     throws ConnectionException;
}
```

All the boiler plate code has to do is call **connect** to get a **Connection** object and **disconnect** to release it when it's done. It does not need to care how the object is created or how, or even if, it is released. This means that **ConnectionPolicy** could even be implemented as a **Connection** pool. The implementation of **ConnectionException** is trivial (Listing 2) and simply there to force a common exception type to be thrown by the policy. It extends **ResouceHandlerException**, which we'll look at later.

Listing 3

```
public abstract class AbstractConnectionPolicy
   implements ConnectionPolicy
{
  @Override
  public void disconnect(final Connection con)
     throws ConnectionException
  {
    try
    {
      if (con != null)
      {
        con.close();
      }
    }
    catch(final SQLException e)
    {
      throw new ConnectionException(
         e.getMessage(), e);
    }
  }
}
```

Listing 4

```
public class ExistingConnection extends
   AbstractConnectionPolicy
{
  private final Connection con;
  private final boolean cleanup;
  public ExistingConnection(final Connection con,
     boolean cleanup)
  {
    this.con = con;
    this.cleanup = cleanup;
  }
  public ExistingConnection(final Connection con)
  {
    this(con,false);
  }
  @Override
  public Connection connect() throws
     ConnectionException
  {
    return con;
  }
  @Override
  public void disconnect(final Connection con)
           throws ConnectionException
  {
    if (cleanup)
    {
      super.disconnect(con);
    }
  }
}
```

Most connections, if they're closed by the policy, will be closed in the same way: by calling **`close`** on the **`Connection`** object. So it's worth putting the common **`close`** code into an abstract class (Listing 3).

All this does is check that the **`Connection`** object is not **`null`**, call **`close`** on it and catch and translate any exception it might throw. Implementing a connection policy for an existing connection is then very simple (Listing4).

A **`Connection`** object is passed in through the constructor and passed back via **`connect`**. An overloaded constructor allows the cleanup flag to be set (by default it is not set). When **`disconnect`** is called the flag is checked to see if the connection should be cleaned up. Although it's simple, it's not as simple as it could be as I have added the cleanup flag, but the policy is more flexible this way.

'Effective Java'

Item 2: Consider a builder when faced with many constructor parameters

The item gives an example of a class that has a number of optional properties that should be initialised via a constructor and explains how they can be initialsed using the telescoping constructor pattern or the JavaBean pattern. It concludes that 'the telescoping constructor pattern works, but it is hard to write client code where there are many parameters, and harder still to read it' and that 'the JavaBean pattern precludes the possibility of making a class immutable'. His argument in both cases is convincing.

As a solution, the item suggests a variation on the builder pattern. Basically, the class with the optional properties has an inner class that can be used to initialise the properties on construction. The resulting initialisation syntax looks like this:

```
NutritionFacts cocaCola =
   new NutritionFacts.Builder(240,8).
   calories(100).sodium(35).carbohydrate(27).
   build();
```

The item points out that 'the builder pattern stimulates named optional parameters' and that the pattern does not really become useful until you have at least four optional properties, and if needed should be used as soon as possible, as refactoring to the pattern can be problematic.

The item summarises the builder pattern as 'a good choice when designing classes whose constructors or static factories would have more than a handful of parameters'.

A policy for creating a connection from a connection string and driver is a little more involved, but the beauty of boiler plate code is that you only have to write it once (Listing 5).

`StringConnection` extends **`AbstractConnectionPolicy`** and takes advantage of the common **`disconnect`** method. The constructor takes a driver string (e.g. **`"sun.jdbc.odbc.JdbcOdbcDriver"`**) and a connection string (e.g. **`"jdbc:odbc:Marauder"`**) and uses them to set the appropriate members. The **`setUser`** and **`setDatabase`** methods are a variation on the builder pattern, as described by Joshua Bloch's second 'Effective Java' item (see sidebar above), that allows a username and password and a default database to be set optionally. For example:

```
final ConnectionPolicy conPolicy
  = new StringConnection(driver,connectionString)
    .setUser(username, password)
    .setDatabase(database);
```

The **`connect`** override uses the **`driver`** and **`connectionString`** objects to create a **`Connection`** object, and catches, translates and rethrows any exceptions. It also calls the **`useDatabase`** method. If the **`database`** object has been set it uses it to set the current database by building the appropriate SQL statement and using a **`Statement`** object to execute it. The **`Statement`** object is cleaned up by a finally block. If an exception is thrown at any point, the **`Connection`** object is closed, but any close exception is ignored in favour of the original exception, and then the original exception is rethrown. This is one of those rare cases where an exception can be ignored, but should be logged.

Resource handler

With the FINALLY FOR EACH RELEASE pattern and the **`ConnectionPolicy`** interface as a starting point it is possible to start building up a class that will handle the cleanup of resources automatically (Listing 6).

The constructor takes and stores a **`ConnectionPolicy`** object. The **`executeQuery`** method uses the policy to create, use and cleanup a **`Connection`** object. A new feature called Generics [8] was introduced in Java 1.5. One of the advantages of generics is that you can specify the type used by a class when an instance of that class is declared. In the case of the **`ResourceHandler`** the return type of the **`executeQuery`** method is paramatized so that it can return any type. As we'll see later, this is useful if the SQL query is returning something other than a single string; such as a list of strings or key value pairs.

Listing 5

```
public class StringConnection extends
   AbstractConnectionPolicy
{
  private final String driver;
  private final String connectionString;
  private String database = null;
  private String username  = null;
  private String password = null;
  public StringConnection(  final String driver,
     final String connectionString)
  {
    this.driver = driver;
    this.connectionString = connectionString;
  }
public StringConnection setUser(
   final String username, final String password)
  {
    this.username = username;
    this.password = password;
    return this;
  }
  public StringConnection setDatabase(
     final String database)
  {
    this.database = database;
    return this;
  }
  @Override
  public Connection connect() throws
     ConnectionException
  {
    Connection con = null;
    try
    {
      Class.forName(driver);
      con = DriverManager.getConnection(
         connectionString, username, password);
      useDatabase(con,database);
    }
    catch(ClassNotFoundException e)
    {
      throw new ConnectionException(
         e.getMessage(),e);
    }
    catch(SQLException e)
    {
      throw new ConnectionException(
         e.getMessage(),e);
    }
    return con;
  }
private void useDatabase(final Connection con,
   final String database) throws SQLException
  {
    if (database != null)
    {
      try
      {
        final Statement stmt =
           con.createStatement();
        try
        {
          final StringBuilder sql
            = new StringBuilder("USE [");
          sql.append(database);
          sql.append("]");
          stmt.execute(sql.toString());
        }
        finally {
          stmt.close();
        }
      }
      catch(final SQLException e1)
      {
        try {
          con.close();
        }
        catch(final SQLException e2) {
          // Swallow
        }
        throw e1;
      }
    }
  }
}
```

Listing 5 (cont'd)

Listing 6

```
public class ResourceHandler<Value>
{
  private final ConnectionPolicy conPolicy;
  public ResourceHandler(
     final ConnectionPolicy conPolicy)
  {
    this.conPolicy = conPolicy;
  }
  public Value executeQuery(final String sql)
     throws ResourceHandlerException
  {
    Value result = null;
    Connection con = conPolicy.connect();
    try
    { // use }
    catch(final SQLException e)
    { // Report Error }
    finally
    {
      try
      {
        conPolicy.disconnect(con);
      }
      catch(final ConnectionException e)
      { // Report Error }
    }
    return result;
  }
}
```

The next step after creating the connection is to create a **`Statement`** object (Listing 7).

A **`Statement`** object is created by the **`createStatement`** method, which is protected so that it can be overridden in a subclass if a different type of statement needs to be created. A **`ResultSet`** object can be created in much the same way (Listing 8). Once the result set has been created, another method is called to create the return value (Listing 9).

The return value will always be **`null`** unless **`getValue`** is overridden in a subclass. Finally a way of reporting errors needs to be implemented. In some circumstances it may be sufficient to send the error to standard out via **`printStackTrace`**. In others, the throwing of an exception may be desirable. The use of a policy allows these and other error reporting methods to be implemented:

```
public interface ErrorPolicy
{
  abstract void handleError(final Exception e)
     throws ResourceHandlerException;
  abstract void handleCloseError(
     final Exception e)
     throws ResourceHandlerException;
}
```

Listing 7

```
public Value executeQuery(final String sql)
   throws ResourceHandlerException
  {
    Value result = null;
    Connection con = conPolicy.connect();
    try
    {
      final Statement stmt =
         createStatement(con);
      try
      { // use statement }
      finally
      {
        try
        {
          stmt.close();
        }
        catch(final SQLException e)
        { // Report Error }
      }
    }
    catch(final SQLException e)
    { // Report Error }
    finally
    {
      try
      {
        conPolicy.disconnect(con);
      }
      catch(final ConnectionException e)
      { // Report Error }
    }
    return result;
  }
  protected Statement createStatement(
     final Connection con) throws SQLException
  {
    return con.createStatement();
  }
```

There is a general error handler, and a handler for errors caused by releasing resources, because a client may wish to handle release errors differently, for example logging them rather than throwing. The **`ResourceHandlerException`** looks like this:

```
public class ResourceHandlerException
   extends Exception
{
  public ResourceHandlerException(String message,
    Throwable throwable)
  {
    super(message, throwable);
  }
}
```

The default error policy ought to throw an exception and should only allow the first exception it receives to be thrown. This is so that the first exception detailing the real problem doesn't get lost when subsequent exceptions are thrown (Listing 10).

Error policies are integrated into **`ResourceHandler`** using another builder method and by calling **`handleError`** or **`handleCloseError`** in the appropriate catch blocks (Listing 11).

Java has another wonderful feature called anonymous classes. In *Java In A Nutshell* [9] David Flanagan describes anonymous classes as:

> ...a local class without a name. Instead of defining a local class and then instantiating it, you can often use an anonymous class to combine these two steps

Listing 8

```
public Value executeQuery(final String sql)
   throws ResourceHandlerException
  {
    Value result = null;
    Connection con = conPolicy.connect();
    try
    {
      final Statement stmt =
         createStatement(con);
      try
      {
        final ResultSet rs =
           getResultSet(stmt,sql);
        try
        { // use }
        finally
        {
          try
          {
            rs.close();
          }
          catch(final SQLException e)
          { // Report Error }
        }
      }
      catch(final SQLException e)
      { // Report Error }
      finally
      {
        try
        {
          stmt.close();
        }
        catch(final SQLException e)
        { // Report Error }
      }

    }
    ...
    return result;
  }
  ...
  protected ResultSet getResultSet(
     final Statement stmt, final String sql)
     throws SQLException
  {
    return stmt.executeQuery(sql);
  }
```

An anonymous class can be used to implement a custom **`getValue`** method. The following example shows how anonymous classes can be used to execute an SQL statement that returns a single string:

```
final String url = new ResourceHandler<String>(
   new StringConnection(driver,connectionString))
  {
    @Override
    protected String getValue(final ResultSet rs)
       throws SQLException
    {
      if (rs.next())
      {
        return rs.getString("url");
      }
      return null;
    }
  }.executeQuery(
     "select url from services where name =
     'Instruments'");
```

Listing 9

```
public Value executeQuery(final String sql)
      throws ResourceHandlerException
  {
    Value result = null;
    Connection con = conPolicy.connect();
    try
    {
      final Statement stmt =
         createStatement(con);
      try
      {
        final ResultSet rs =
           getResultSet(stmt,sql);
        try
        {
          result = getValue(rs);
        }
        catch(SQLException e)
        { // report error }
        finally
        {
          try
          {
            rs.close();
          }
          catch(final SQLException e)
          { // report error }
        }
      }
      catch(final SQLException e)
        { // report error }
      finally
      {
        try
        {
          stmt.close();
        }
        catch(final SQLException e)
        { // report error }
      }

    }
  ...
  }
  protected Value getValue(final ResultSet rs)
     throws SQLException
  {
    return null;
  }
```

Alternatively if multiple strings are required:

```
final List<String> urls =
   new ResourceHandler<List<String>> (
   new StringConnection(driver,connectionString))
  {
    @Override
    protected List<String> getValue(
       final ResultSet rs) throws SQLException
    {
      List<String> result =
         new ArrayList<String>();
      while(rs.next())
      {
        result.add(rs.getString("url"));
      }
      return result;
    }
  }.executeQuery("select url from services");
```

Listing 10

```
public class ThrowOnError implements ErrorPolicy
{
  private Exception exception = null;
  @Override
  public void handleError(Exception e) throws
     ResourceHandlerException
  {
    if (exception == null)
    {
      exception = e;
      throw new ResourceHandlerException(
         e.getMessage(),e);
    }
  }
  @Override
  public void handleCloseError(Exception e)
     throws ResourceHandlerException
  {
    handleError(e);
  }
}
```

Listing 11

```
public class ResourceHandler<Value>
{
  private final ConnectionPolicy conPolicy;
  private ErrorPolicy errorPolicy =
     new ThrowOnError();
  ...
  public ResourceHandler<Value> setErrorPolicy(
     final ErrorPolicy errorPolicy)
  {
    this.errorPolicy = errorPolicy;
    return this;
  }
  public Value executeQuery(final String sql)
     throws ResourceHandlerException
  {
    Value result = null;
    Connection con = conPolicy.connect();
    try
    {
      final Statement stmt =
         createStatement(con);
      try
      {
        final ResultSet rs =
           getResultSet(stmt,sql);
        try
        {
          result = getValue(rs);
        }
        catch(SQLException e)
        {
          errorPolicy.handleError(e);
        }
        finally
        {
          try
          {
            rs.close();
          }
          catch(final SQLException e)
            {
              errorPolicy.handleCloseError(e);
            }
          }
        }
        catch(final SQLException e)
        {
```

Listing 11 (cont'd)

```
        errorPolicy.handleError(e);
      }
      finally
      {
        try
        {
          stmt.close();
        }
        catch(final SQLException e)
        {
          errorPolicy.handleCloseError(e);
        }
      }
    }
    catch(final SQLException e)
    {
      errorPolicy.handleError(e);
    }
    finally
    {
      try
      {
        conPolicy.disconnect(con);
      }
      catch(final ConnectionException e)
      {
        errorPolicy.handleCloseError(e);
      }
    }
    return result;
  }
...
}
```

Executing statements that do not return a resultSet

The **ResourceHandler** works fine until you do something like this:

```
new ResourceHandler(conPolicy).executeQuery(
  "insert into services ([Name],[url])
  values('Engine','http://prodserv01/axis/
  services/Engine')");
```

The above statement doesn't return a result set and unhelpfully Java throws an exception to let you know:

```
ResourceHandlerException: No ResultSet was produced
```

The best way to get around this is to add an **execute** method, similar to the **executeQuery** method, but without a return value (Listing 12).

Finally, if you need to execute multiple statements that do not return result sets, you can do this:

```
new ResourceHandler(conPolicy).execute("..");
new ResourceHandler(conPolicy).execute("..");
new ResourceHandler(conPolicy).execute("..");
new ResourceHandler(conPolicy).execute("..");
new ResourceHandler(conPolicy).execute("..");
```

However, with the **StringConnection** policy this would create and destroy the connection for each statement call. The alternative is to modify execute to take an array of statements and iterate through them creating a **Statement** object for each (Listing 13).

This has the draw back that even if you have just a single statement to execute, it has to be passed in as an element of an array. This can be easily overcome by adding an execute overload:

```
public void execute(final String sql)
   throws ResourceHandlerException
{
  execute(new String[]{sql});
}
```

Listing 12

```
public void execute(final String sql)
   throws ResourceHandlerException
{
  Connection con = conPolicy.connect();
  try
  {
    final Statement stmt = createStatement(con);
    try
    {
      execute(stmt,sql);
    }
    catch(final SQLException e)
    {
      errorPolicy.handleError(e);
    }
    finally
    {
      try
      {
        stmt.close();
      }
      catch(final SQLException e)
      {
        errorPolicy.handleCloseError(e);
      }
    }
  }
  catch(final SQLException e)
  {
    errorPolicy.handleError(e);
  }
  finally
  {
    try
    {
      conPolicy.disconnect(con);
    }
    catch(final ConnectionException e)
    {
      errorPolicy.handleCloseError(e);
    }
  }
}
...
protected void execute(final Statement stmt,
   final String sql) throws SQLException
  {
    stmt.execute(sql);
  }
```

Conclusion

I was reviewing some code where I work recently and I came across the code in Listing 14.

I summoned my two co-developers over, one of whom had written it, and asked them to find the problem. They looked at it for a good 90 seconds before I started offering clues. As soon as I asked what would happen if **rs.close()** threw, the penny dropped and the lights went on. The answer of course is that the **Statement** and **Connection** objects would not get closed. The real break through came when one commented that Java programs must be failing to close database connections properly all over the place. The way Java handles resource cleanup is ludicrously verbose and the idea of a method throwing when a resource is closed just plain ridiculous.

However, I believe that I have shown here that this can be overcome using FINALLY FOR EACH RELEASE, and the amount of code needed to query a database can be reduced significantly with use of some simple boiler plate.

In part II I will look at another possible solution using EXECUTE AROUND METHOD. ■

Listing 13

```
public void execute(final String[] sql)
   throws ResourceHandlerException
  {
    Connection con = conPolicy.connect();
    try
    {
      for(final String s : sql)
      {
        final Statement stmt =
           createStatement(con);
        try
        {
          execute(stmt,s);
        }
        catch(final SQLException e)
        {
          errorPolicy.handleError(e);
        }
        finally
        {
          try
          {
            stmt.close();
          }
          catch(final SQLException e)
          {
            errorPolicy.handleCloseError(e);
          }
        }
      }
    }
    catch(final SQLException e)
    {
      errorPolicy.handleError(e);
    }
    finally
    {
      try
      {
        conPolicy.disconnect(con);
      }
      catch(final ConnectionException e)
      {
        errorPolicy.handleCloseError(e);
      }
    }
  }
```

Listing 14

```
...
finally
  {
    try
    {
      if (rs != null)
      {
        rs.close();
      }
      if (stmt != null)
      {
        stmt.close();
      }
      if (con != null)
      {
        con.close();
      }
    }
    catch(SQLException e)
    {
      // Translate exception
    }
  }
```

Acknowledgements

Thank you to Tony Barret-Powell, Kevlin Henney, Adrian Fagg and Russel Winder for advice and guidance (and for not spotting the huge flaw in my original proposal! ;-)).

References

[1] *Effective Java: A Programming Language Guide* by Joshua Bloch. ISBN-13: 978-0321356680

[2] C# IDisposable interface: http://msdn.microsoft.com/en-us/library/system.idisposable.aspx

[3 'Handling Exceptions in Finally' by Tony Barrett-Powell: http://accu.org/index.php/journals/236

[4] 'Exceptional Java' by Alan Griffiths: http://accu.org/index.php/journals/399

[5] 'More Exceptional Java' by Alan Griffiths: http://accu.org/index.php/journals/406

[6] 'Java Generics': http://java.sun.com/j2se/1.5.0/docs/guide/language/generics.html

[7] Sun Java Docs: http://java.sun.com/reference/docs/

[8] Another Tail of Two Patterns: http://www.two-sdg.demon.co.uk/curbralan/papers/AnotherTaleOfTwoPatterns.pdf

[9] *Java in a Nutshell* by David Flanagan. ISBN-13: 978-0596007737

Regaining Control Over Objects Through Constructor Hiding

Mike Crow describes an alternative form of object construction.

Construction often feels like one of the least flexible aspects of the interface a C++ class provides to its client code. At the time of traditional construction, the client must specify the exact type of the object to be created, gets to decide where that creation occurs and effectively has a veto over how the object can manage its lifetime. This article explains how hiding the constructor and providing a separate means to create the object gives this control back to the class itself.

The technique

Just because a class must provide a constructor is no reason for it to be generally available. It is quite common to make the **copy** constructor private [1] in order to stop client code from accidentally copying an instance. If the default constructor is also made private then only the class itself or its friends can create instances.

Listing 1 demonstrates a class which allows instances to be created using a static create function (also known as a 'factory function'). Beyond that it doesn't do anything useful. Even with this simple example we have ensured that clients cannot create instances on the stack.

Listing 1

```
class HiddenConstructor
{
private:
  HiddenConstructor() {}
  HiddenConstructor(const HiddenConstructor &);
public:
  static HiddenConstructor *Create()
  {
    return new HiddenConstructor;
  }
};

int main()
{
  HiddenConstructor *obj =
     HiddenConstructor::Create();
  delete obj;
  return 0;
}
```

Controlling the return type

Once you have forced clients to call a function to create the object then you also have control over the return type of that function. Smart pointers such as **boost::shared_ptr** [2] and **tr1::shared_ptr** [3] offer automatic management of object lifetimes, but in general they rely on every part of the code using the same smart pointer implementation for that particular class. In particular, the creator of the object should immediately assign it to a smart pointer. Listing 2 shows how a static create function allows this behaviour to be enforced by the class itself, thus ensuring the raw pointer is not exposed, and removing the risk of it being used directly (the smart pointer may provide a means to extract the underlying pointer but anyone using it should hopefully realise they are on shaky ground). The class can even assume that it is being represented by a shared pointer if it needs to pass pointers to itself to other code.

Listing 2

```
#include <boost/shared_ptr.hpp>
class SmartObject;
typedef boost::shared_ptr<SmartObject>
   SmartObjectPtr;
class SmartObject
{
  private:
    SmartObject() {}
    SmartObject(const SmartObject &);
    void operator=(const SmartObject &);
  public:
    static SmartObjectPtr Create()
    {
      return SmartObjectPtr(new SmartObject);
    }
};

int main()
{
  SmartObjectPtr obj = SmartObject::Create();
  return 0;
}
```

Creation failure

Dealing with failure during construction in C++ has traditionally been problematic. In theory, exceptions solve this problem, but there are still some situations where, whether due to inadequate tools or system constraints, exceptions are not applicable. Being able to control the exact signature of the create function means that construction failure can be indicated in a flexible way. The code in the class itself may not be pretty but at least it can be encapsulated. Listing 3 shows an example of this technique.

Implementation hiding

Pimpl [4] is a popular technique for separating interface and implementation. It hides the implementation innards of a class from its clients and decreases coupling by replacing the data members of a class with a single pointer to a forward declared structure. This structure is declared in the implementation file that contains the data members, and all data member accesses need to be made through the pointer.

Listing 4 shows an alternative that provides an abstract interface to clients using pure virtual functions and a static create function to create a concrete instance that actually implements all the functions. No forwarding functions are required and no special work needs to be done to access data members. There is the slight penalty of a virtual call for each function call.

This method can't completely replace Pimpl since it is much harder for client code to derive from such a class. However, code that is part of the implementation can derive from the class which gives rise to other techniques as will be seen later.

MIKE CROWE

Mike Crowe started a short-term contract ten years ago as a Windows developer and ended up writing embedded Linux software too and found it much more fun. Contact Mike at: mac@mcrowe.com

Listing 3

```
class FailingCreation
{
  public:
    enum Error
    {
      SUCCESS = 0, OUT_OF_MEMORY = 1,
      INSUFFICIENT_FROBNICATORS = 2
    };
  private:
    FailingCreation() {}
    FailingCreation(const FailingCreation &);
    Error Init()
    {
      // Do more complex initialisation and
      // return zero to indicate success or
      // non-zero to indicate failure.
      return SUCCESS;
    }

public:
  static Error Create(FailingCreation **result)
  {
    FailingCreation *p = new FailingCreation;
    if (!p)
    return OUT_OF_MEMORY;
    Error error = p->Init();
    if (error != SUCCESS)
      delete p;
    else
      *result = p;
      return error;
  }
};

int main()
{
  FailingCreation *obj;
  if (FailingCreation::Create(&obj) ==
     FailingCreation::SUCCESS)
  {
    // Use object
    delete obj;
  }
  else
  {
    // Failed to create
  }
  return 0;
}
```

One interface, many implementations

There is no requirement that a static create function always creates the same type of object every time it is called. It just needs to create something that implements the correct interface. The exact type of the object created could be decided at compile time or it could also depend on parameters passed to the factory function. Listing 5 shows a combination of both methods used to create an audio decoder that's appropriate for the platform and the type of file being played. This method hides away the preprocessor magic in a single more manageable place without polluting the client code.

Lifetime control

In Listing 1 we saw how we could stop an instance being created on the stack. This means that the client can code no longer (easily) decide when the object is actually deleted. For example, if the instance has a window associated with it, we might want to wait until the window has been closed before the object is destroyed, as long as we can guarantee that this will happen before the program exits. In this case you may want to make the destructor private too.

Listing 4

```
// Header (exposed to clients)
#include <boost/shared_ptr.hpp>
class Interface;
typedef boost::shared_ptr<Interface>
InterfacePtr;
class Interface
{
protected:
  Interface() {}
public:
  virtual ~Interface() {}
  virtual void MemberFunction(
     const char *data) = 0;
  static InterfacePtr Create();
};
// Implementation (hidden from clients)
#include <string>
class Implementation : public Interface
{
  std::string m_data;
public:
  virtual void MemberFunction(const char *data);
};
InterfacePtr Interface::Create()
{
  return InterfacePtr(new Implementation);
}
void Implementation::MemberFunction(
   const char *data)
{
  m_data = data;
}
```

Conclusion

Although the technique I describe is not revolutionary I believe that it can help to make code cleaner and safer with very little effort. I've used it in all the situations described with great success. ■

Thanks

Thanks to Pete Goodliffe for reviewing drafts of this article.

References

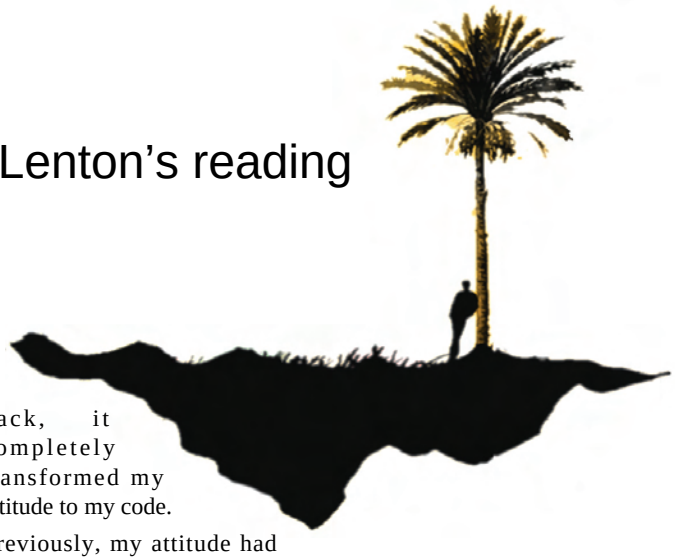
[1] CERT C++ Secure Coding Practices MEM41-CPP. http://tinyurl.com/6xte5h
[2] http://www.boost.org/
[3] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1450.html
[4] http://c2.com/cgi/wiki?PimplIdiom (This page actually goes on to describe something very similar to the method I've described here.)

Listing 5

```
#include <boost/shared_ptr.hpp>
#include <stdint.h>
class AudioDecoder;
typedef boost::shared_ptr<AudioDecoder>
AudioDecoderPtr;
class AudioDecoder
{
  AudioDecoder();
public:
  static AudioDecoderPtr Create(
     const std::string &filename);
  virtual ~AudioDecoder();
  virtual void ReadSamples(uint16_t *samples,
     unsigned int sample_count) = 0;
};
```

Desert Island Books

Paul Grenyer introduces Alan Lenton's reading selection.



It's difficult to be concise about Alan Lenton as there is so much to say. I first met him one Tuesday afternoon in the Dirty Duck in Leamington Spa (you know, that conference at the car museum). He was there for the now legendary Phil Hibbs opener 'I know someone who's looking forward to disliking you....' and I haven't been able to shake him since.

Over the years Alan and I have been regular drinking partners at all the ACCU conferences we have both attended as well as all the London pub meets. Alan has always been an enthusiastic and committed contributor to the ACCU Mentored Developers. Alan brings with him a certain charisma, and even saved me from organising a pub meet in Kings Cross once....

Alan Lenton

Coming after such luminaries as Paul Grenyer and Jez Higgins is a difficult task indeed, but I will do my best to follow in their footsteps. [Paul, is that enough to get me in to CVu, or do I need to lay it on a bit thicker?]

A desert island... Hmmm...

This is a bit alarming given the rise in sea levels due to global warming. I hope I get rescued before the rising sea level covers my island! (Try living in Norwich! – Paul).

So, apart from an unlimited supply of Bombay Sapphire Gin & Tonic, what would I take with me?

Programming books...

Programming books, and in my case one stands out above all others: Martin Fowler's *Refactoring* [1]. When I first read this book, which I picked up at the Blackwells stall at the ACCU Conference a few years back, it completely transformed my attitude to my code. Previously, my attitude had been one of 'if it ain't broke, don't fix it'. Tampering with working code when I first started programming taught me a few unpleasant lessons about how dangerous it could be. Martin Fowler's book taught me that it is possible to improve the quality of existing code without breaking it, and explained clearly and concisely just how to do it safely. As a result the quality of code that I now write has greatly improved, and I think that I'm a much better programmer for it.

The book was also my first introduction to the world of Agile, which I embraced with all the enthusiasm of the convert. Not surprising really; the books written by the founders of the agile movement are, in general, highly readable and filled with enthusiasm. I take what I consider to be a more balanced view now, but I retain a healthy respect for agile methods.

That was the easy choice; the other choices are more difficult, because there is a whole slew of books that I could choose. Some of them are obvious – Nico's *Standard Template Library* [2], for instance, but Paul covered that a couple of issues ago. No. I'm looking for something a little different, and one that stood out is a book Bjarne Stroustrup wrote fourteen years ago, *The Design and Evolution of C++* [3]. At the time I read it I

Regaining control over objects through constructor hiding (continued)

Listing 5 (cont'd)

```
class GenericMp3Decoder : public AudioDecoder
{
public:
  GenericMp3Decoder(const std::string &filename);
  virtual void ReadSamples(uint16_t *,
     unsigned int);
  static bool CanPlay(
   const std::string &filename);
};

#if defined(__i386__)
class OptimisedX86Mp3Decoder :
   public AudioDecoder
{
public:
  OptimisedX86Mp3Decoder(
     const std::string &filename);
  virtual void ReadSamples(uint16_t *,
     unsigned int);
  static bool CanPlay(
     const std::string &filename);
};
#endif // defined(__x86__)
class WavDecoder : public AudioDecoder
{
```

Listing 5 (cont'd)

```
public:
  WavDecoder(const std::string &filename);
  virtual void ReadSamples(uint16_t *,
     unsigned int);
  static bool CanPlay(
     const std::string &filename);
};
AudioDecoderPtr AudioDecoder::Create(
   const std::string &filename)
{
  if (WavDecoder::CanPlay(filename))
  return AudioDecoderPtr(
     new WavDecoder(filename));
#if defined(__i386__)
  else if (
     OptimisedX86Mp3Decoder::CanPlay(filename))
  return AudioDecoderPtr(
     new OptimisedX86Mp3Decoder(filename));
#endif // defined(__x86__)
  else if (GenericMp3Decoder::CanPlay(filename))
  return AudioDecoderPtr(
     new GenericMp3Decoder(filename));
  else
  return AudioDecoderPtr();
}
```

was still tending to use C++ as merely a better C (and it is better, by the way!). The book helped me understand what C++ was really about, and at that time I definitely needed the help!

As the legendary Verity Stob once put it, 'Whereas smaller computer languages have features designed into them, C++ is unusual in having a whole swathe of functionality discovered, like a tract of 19th century Africa. Nobody intended that C++ should support template metaprogramming, a wildly clever but visually dismal technique that allows the programmer to write a program within the program. One day it was noticed that it could be done, and now it is done.' [4]

Wise words from a talented lady!

My third choice of book is not, in fact, a programming book at all. It's by Robert G Williams, and it's called *The Money Changers* [5]. It is a tour of the global money markets featuring interviews with the people involved in international currency exchanges. If you've ever been curious about what happens behind the scenes when you stick your credit card in an ATM in (say) Ulan Bator, then this is the book for you.

I bought it because I needed to understand money markets for an enhancement I was planning for my online space trading game, Federation 2 [6]. The book was a fabulous read, and for the first time I understood what was happening in the money markets. The result was that I put my plans for currency markets, where each planet had its own currency, on hold, and left it with just commodity trading, commodity futures trading, and company stock trading.

One day I will implement currency markets, but it's just too big a project for now! In the meantime the ideas the book taught me have enabled me to understand the current financial crisis, and figure out what it is that the central banks are trying to do.

My final non-fiction book is *Unix Network Programming*, *Volume 1* by W. Richard Stevens [7]. It's known in the trade as simply 'Stevens', and it's the classic book on network programming. I learned how to do network programming from the first edition, and by the time I recently replaced it, it was much thumbed and suffering from brown, brittle pages.

This book has everything a programmer needs to know about networks, including design of the networking part of both servers and clients. While a lot of frameworks and high level libraries hide the implementation details (which this book also covers), it remains important to understand what a library is doing if you want to use it efficiently. Network programming is, and has always been, one of those areas where efficiency and competence matter. Oh, and contrary to the title, it's about more than just Unix programming.

...Fiction...

This was a straight fight between William Gibson's *Neuromancer* [8] and Neal Stephenson's *Snow Crash* [9] – the two defining books of the cyberpunk milieu. Eventually I settled on *Snow Crash* which has more substance. It's set in a world where governments have lost control of their tax revenues, the mafia run pizza delivery services, and the whole net is a virtual world. Our hero is Hiro Protagonist, a hacker, samurai swordsman and pizza-delivery driver.

I'm not going to tell you the story, but it's fabulous – buy the book and read it for yourself over the summer holiday.

...Two CDs...

It's a difficult decision. I have hundreds of CDs to choose from. Paul already grabbed Marillion's *Misplaced Childhood* [10], so that's off. I think one of the CDs has to be Pink Floyd's *Dark Side of the Moon* [11]. I first heard it on a sunny afternoon while at college, and it has haunted me ever since, both for the music and for the memories it evokes.

My second CD was a toss up between Tom Paxton's *The Compleat Tom Paxton* [12], and *Waiting for Bonaparte* by Men They Couldn't Hang [13].

Eventually Tom Paxton won by a whisker. I regard Tom Paxton as one of the best US folk singer/songwriters and this is a live double CD. The CD showcases not just Paxton's songs but also his story telling – 'Talking Vietnam Pot Luck Blues' has to be heard to be believed.

...And a DVD

Yet more decisions – I guess the 40 DVD set of everything to do with *Babylon 5* [14] is probably cheating. I think I'll plump for *Gettysburg* starring Martin Sheen as Robert E Lee [15]. I was lucky enough to be taken on a tour of the Gettysburg battlefield by a friend who had studied the battle at US marine staff college. Having seen the terrain – and in the US, unlike Britain, they preserve their Civil War heritage – I found the film all the more remarkable. Viewing it for the first time I started to grasp the strategy, tactics and decision making of both sides. And the tragedy of friends killing one another. A truly remarkable movie – not to mention some of the most fantastic beards ever seen on film!

Well that about wraps it up. All I need now to complete my collection is a generator, a very large supply of fuel and a laptop :)

References

[1] *Refactoring* by Martin Fowler, Addison-Wesley, ISBN 0-201-48567-2

[2] *The C++ Standard Library* by Nicolai M. Josuttis. Addison-Wesley ISBN 0-201-37926-0

[3] *The Design and Evolution of C++* by Bjarne Stroustrup. Addison-Wesley ISBN 0-201-54330-3

[4] http://www.regdeveloper.co.uk/2006/05/05/cplusplus_cli/

[5] *The Money Changers* by Robert G Williams. Zed Books ISBN 1-84277-095-9

[6] http://www.ibgames.com

[7] *Unix Network Programming, Volume 1*, 3rd Edition by W Richard Stevens, Bill Fenner and Andrew M Rudoff. Addison-Wesley ISBN 0-13-141155-1

[8] *Neuromancer* by William Gibson. Voyager ISBN 0-006-48041-1

[9] *Snow Crash* by Neal Stephenson. Penguin Books ISBN 0-140-23292-3

[10] *Misplaced Childhood* by Marillion. EMI

[11] *Dark Side of the Moon* by Pink Floyd. EMI

[12] *The Compleat Tom Paxton* by Tom Paxton. Elektra

[13] *Waiting for Bonaparte* by Men They Couldn't Hang. Magnet

[14] *Babylon 5: The Complete Collection + The Lost Tales.* Warner Home Video

[15] *Gettysburg*. Warner Home Video

What's it all about?

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

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

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

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

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

Next issue: Ian Bruntlett picks his desert island books.

Code Critique Competition 55

Set and collated by Roger Orr.

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

Last issue's code

I'm trying to write a simple logging header file, but it doesn't seem to be doing the right thing – my errors aren't being logged. Can someone help me sort this out?

(Apologies for the missing "\" on the email version of the critique)

Listing 1

```
// ---- logging.h -----
enum { ERROR, WARN, DEBUG };

int level;

std::string now()
{
  time_t timeNow = time( 0 );
  char buffer[ 20 ];
  strftime( buffer, sizeof( buffer ),
    "%d %b %H:%M:%S", localtime( &timeNow ) );
  return buffer;
}

#define LOG(LEV, X) \
{ \
  if ( level & LEV ) { \
    std::ostringstream oss; \
    oss << now() << " [" << #LEV << "] " \
        << (X) << std::endl; \
    printf( oss.str().c_str() ); \
  } \
}

#define LOG_ERROR(x) \
LOG(ERROR, x)

#define LOG_WARN(x) \
LOG(WARN, x)

#define LOG_DEBUG(x) \
LOG(DEBUG, x)

// ---- example.cpp ----
#include <ctime>
#include <iostream>
#include <string>
#include <sstream>

#include "logging.h"

int main()
{
  level = DEBUG | WARN | ERROR;
  LOG_DEBUG( "This is a test" );

  LOG_WARN( "This is a warning" );

  std::ostringstream oss;
  oss << "An example msg";

  LOG_ERROR( "Problem: " + oss.str() );
}
```

Critiques

Ian Bruntlett <ianbruntlett@hotmail.com>

It's been a while since I've used i/o streams. Here are the changes I made to the student code critique to get it to work.

```
// logging.h line 2
enum { ERROR=1, WARN=2, DEBUG=4 };
// needed because default values not right

// logging.h line 20 added \ to end of line in
// order to compile
```

Later on in `example.cpp` - I basically muddled around until it worked. I would be interested in seeing an article in CVu/Overload about this kind of thing.

```
std::ostringstream oss;
oss << "An example msg";
std::string msg("Problem: ");
msg += oss.str();
LOG_ERROR(msg.c_str());
```

Ken Duffill <k.duffill@ntlworld.com>

In the beginning...

I copied the two files Logging.h and example.cpp to my two build environments. Visual Studio 2008 Express (which is also configured to run PC-Lint over the code) and gcc under cygwin.

Both are configured to return maximum warnings.

When we run PC-Lint over this code it tells us the first four issues we need to deal with.

1. Ignoring return value of function '**`strftime(char *, unsigned int, const char *, const struct tm *)`**'

   It is not a good idea to ignore the return value from **`strftime`** (or any other function for that matter) - it contains valuable information. If **`buffer`** is not big enough for the output string and its terminating nul, **`strftime`** will return 0, and the contents of **`buffer`** will not be defined and may not be nul terminated. So, it is good practice to at least assert that the return from **`strftime`** is not zero.

```
size_t numChars = strftime( buffer, sizeof(
    buffer ), "%d %b %H:%M:%S", localtime(
    &timeNow ) );
assert((
```


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

```
0 != numChars ) && "Check size of buffer");
```

Be careful, though, to assign the return from **strftime** to a variable and then assert that the value is not zero. If you just wrap the call to **strftime** in the **assert** then **strftime** won't get called at all in a release build.

Now, at least, if you add (say) **%Y** into your format string and **strftime** can no longer fit the result and its terminating nul into **buffer** you will get told to increase the size of buffer.

2. Boolean within 'if' always evaluates to **false**

   This is referring to the **if (level & LEV)** in the **LOG** macro (and this one is actually the cause of your [first] problem).

   First we need to understand that

   ```
   enum { ERROR, WARN, DEBUG };
   ```

   will result in the value of **ERROR** being **0**, the value of **WARN** being **1** and the value of **DEBUG** being **2**.

   Now we can see that in **LOG_ERROR**, when **LEV** is **ERROR**, this conditional will always be **false** no matter what the value of level. It appears that level is expected to be a collection of bits each indicating one of the available logging levels. In this case the values of the enums must each have one (and only one) bit set in order for this to work as you expect.

   So, if you change the enum to

   ```
   enum { ERROR = 1, WARN = 2, DEBUG = 4 };
   ```

   this conditional will always correctly determine whether to log according to **LEV**.

3. Declaration of symbol **oss** hides symbol **oss**

   This problem is due to the conflict between the **std::ostringstream oss** declared within the macro **LOG** and the **std::ostringstream oss** declared in **main**. The nature of macros make these conflicts very hard to spot, which is one of their serious flaws.

   So one 'quick fix' is to rename the **std::ostringstream oss** in **main** to **std::ostringstream oss1** and your example now works as it should.

   Alternatively, the whole clause:

   ```
   std::ostringstream oss; \
   oss << now() << " [" << #LEV << "] " \
       << (X) << std::endl; \
   printf( oss.str().c_str() ); \
   ```

   could be replaced by one that uses **std::cout** directly, and some might say more readably:

   ```
   std::cout << now() << " [" << #LEV << "]" \
   << (X) << std::endl; \
   ```

   As we are using C++ here, there really is no reason to use the C Standard I/O directly (i.e. **printf**). We have used the iostream formatting capabilities, and **std::cout** does the job so why not use it. Additionally, you could later make it so that the stream is passed in somehow so you can elect to use **std::cout**, **std::cerr** or **std::clog** (or even some output logging file of your choosing) without changing the logging code at all, something you can't do if you are using **printf**.

4. function **main(void)** should return a value

   It is legal to 'forget' the return in **main** (the only function in the language that will put a **return 0** in for you). However, this is allowed only in order to support legacy code. It is not a good habit to get into. Add a **return 0;** explicitly and everybody (including PC-Lint) knows that you mean 'everything ran OK'.

Now we have got the code past Lint, we can compile it.

Yes I really do mean that. If you use a static checker (and you should) then configure your build system to run it before it runs the compiler. There is no point in compiling 'wrong' code, and if it fails its static checks it is wrong!

gcc under cygwin compiles without warnings. But Visual Studio whines that **localtime** is deprecated and suggests we use **localtime_s** instead. But, as **localtime_s** is not standard, we have to live with **localtime** for now. This is OK so long as we are sure that **timeNow** is a valid **time_t**, which in our code it is, and we are not running in a multi-threaded environment.

So now the code Lints, builds (cleanly, if we have disabled Microsoft's **localtime** is deprecated warning) and when run produces the output we expect.

Hurrah !

But there are other flaws in this example that need to be dealt with lest they become important issues when your program grows.

I assume it is your intention to include `Logging.h` from multiple source files. Unfortunately, with this implementation you can't. To demonstrate this I created a second .cpp file (`second.cpp`) thus:

```
#include "Logging.h"
#include <iostream>
#include <string>
void Test()
{
  LOG_DEBUG( "This is a second test" );
  LOG_WARN( "This is a second warning" );
  std::string s1("A second example msg");
  LOG_ERROR( "Problem: " + s1 );
}
```

The application now fails to build because the function **std::string now()** and the variable **int level** have external linkage and must only be declared once in the program. A better approach is to put **now** and **level** into a class along with the enum, make **level** a private member and initialise it in the constructor (but refusing to provide a default) we then make sure that the caller knows what he is getting when he is using it. Note here that PC-Lint will complain that we have not declared a default constructor. This is actually what we want. We don't want to be able to forget to say what our logging level should be. If you use a static checker (and you should) then you should be able to tell it to be quiet about this warning (but only do that for this class).

Each module that uses **Log** must create its own **Log** object (and give it a **level** parameter) this provides us with the bonus that different modules can have different logging levels if they want to, but we could have decided to pass a (const) reference to the **Log** object created in **main** if that had better suited out design. We then need to refactor the macros into member functions (which is a good thing anyway, remember – macros are BAD) the only down side to this is we cannot use the little trick of getting the name of the parameter by using **#LEV** anymore, but it is a small price to pay. And, of course refactor the code in `example.cpp` and `second.cpp` to use the new class and its member functions.

Now lets look at that enum again. It is common practice to use all uppercase letters for macro names and other **#defines**, before we put the enum into a class there was a potential problem here if some header, included after `Logging.h`, #defined **ERROR** or **WARN** or **DEBUG** to any integer value then the compiler will not complain; it would have just used the defined value when you may have been expecting to use the enum. This problem has now magically gone away because whenever you use the enum it has to be qualified with the class name, so if **ERROR** (or **WARN** or **DEBUG**) get defined, then **Log::ERROR** will become (say) **Log::21**, which won't compile. Other people's libraries can still break your code, but at least not silently. To make it less likely that other people's code would break yours you could change **ERROR** to a more unique name (using both upper and lower case) such as **ERROR_enum** or **ErrorLevel**.

Now that we have done this let's create `Logging.cpp` to hide all the implementation detail of the **`Logging`** class.

Finally, it is important to make header files self-sufficient, and to depend only upon includes needed for the interface so I have moved the

```
#include <string>
```

from `example.cpp` into `Logging.h` and

```
#include <ctime>
#include <iostream>
```

from `example.cpp` into `Logging.cpp`.

Just in case `Logging.h` accidentally gets included more than one in the same module I have added internal include guards

```
#ifndef LOGGING_H
#define LOGGING_H
...
#endif
```

So my files now look like this:

```
// Logging.h
#ifndef LOGGING_H
#define LOGGING_H
#include <string>

class Log
{
public:
  enum {ERROR = 1, WARN = 2, DEBUG = 4};

  //lint -esym(1712, Log)
  Log(int requiredLevel);

  void Debug(std::string const &txt) const;
  void Warn(std::string const &txt) const;
  void Error(std::string const &txt) const;

private:
  int level;

  void Output(std::string const &levelName,
      std::string const &txt) const;
  std::string now() const;
};
#endif


// Logging.cpp
#include "Logging.h"

#include <cassert>
#include <ctime>
#include <iostream>

Log::Log(int requiredLevel)
: level(requiredLevel){}

void Log::Debug(std::string const &txt) const
{
  if (level & DEBUG)
  {
    Output("DEBUG", txt);
  }
}

void Log::Warn(std::string const &txt) const
{
  if (level & WARN)
  {
    Output("WARN", txt);
  }
}

void Log::Error(std::string const &txt) const
{
  if (level & ERROR)
  {
    Output("ERROR", txt);
  }
}

void Log::Output(std::string const &levelName,
  std::string const &txt) const {
  std::cout << now() << " [" << levelName
    << "] " << txt << std::endl; }

std::string Log::now() const
{
  time_t timeNow = time( 0 );
  char buffer[ 20 ] = {0};
  size_t numChars = strftime( buffer,
    sizeof( buffer ), "%d %b %H:%M:%S",
    localtime( &timeNow ) );
  assert(( 0 != numChars ) &&
    "Check size of buffer");
  return buffer;
}


// Second.cpp
#include "Logging.h"
#include <string>
void Test(void)
{
  Log log(Log::ERROR| Log::WARN | Log::DEBUG);
  log.Debug( "This is a second test" );
  log.Warn( "This is a second warning" );
  std::string s1("A second example msg");
  log.Error( "Problem: " + s1 );
}


// Example.cpp
#include "logging.h"
#include <sstream>
void Test(void);
int main()
{
  Log log(Log::ERROR| Log::WARN | Log::DEBUG);
  log.Debug( "This is a test" );
  log.Warn( "This is a warning" );

  std::ostringstream oss;
  oss << "An example msg";

  log.Error( "Problem: " + oss.str() );
  Test();
  return 0;
}
```

Notes:

I **`#include "Logging.h"`** as the first include in `Logging.c`; this helps to ensure that `Logging.h` is self-sufficient. I use **`|`** rather than **`+`** when initialising log, this just helps to show that these are bit fields not just integers. I have included the prototype for Test in `example.cpp` as opposed to creating a new file `Second.h`, this is definitely NOT recommended in real code. We could define the **`Debug`**, **`Warn`** and **`Error`** member functions to be inline. This might save the function call overhead at the expense of increased code size (remember the compiler is not obliged

to inline just because we ask it to), this is a decision to be made by the designer.

Robert Jones <robertgbjones@gmail.com>

The first error in the code is that the logging level enumeration is using the default assigned values, which will give

```
ERROR = 0
WARN = 1
DEBUG = 2
```

which as bit patterns is 00, 01, 10, which is probably not what was intended.

When the **LOG** macro is used with the **ERROR** level, the bitwise test level **& ERROR** will result in **0**, ie false, so errors will never be output.

Using an enumeration definition of

```
enum {ERROR= 0x01, WARN = 0x02, DEBUG = 0x04};
```

will make that aspect of the code work as intended.

With that fixed the second error is uncovered, which is that the extension of the **ERROR** log message (**oss.str()**), is not output.

The problem here is that the macro mechanism has no understanding of C++ syntax and scoping, and is just a dumb text substitution mechanism.

Adapting the **main()** sequence to include the dumb textual substitution performed by the macro gives this code:

```
int main()
{
  level = DEBUG | WARN | ERROR;
  LOG_DEBUG( "This is a test" );

  LOG_WARN( "This is a warning" );

  std::ostringstream oss;
  oss << "An example msg";
  {
    if ( level & DEBUG ) {
      std::ostringstream oss;
      oss << now() << " [" << "DEBUG" << "] "
          << ("Problem: " + oss.str())
          << std::endl;
      printf( oss.str().c_str() );
  }
 }
}
```

From this the issue becomes much clearer. The **oss** in the **main()** is being hidden by the **oss** in the scope of the macro expansion. In general this is a difficult problem to avoid, and an excellent motivation not to use macros.

As a quick fix in this critique, renaming either of the **oss**'s will give the intended results.

Simon Sebright <simonsebright@hotmail.com>

This little piece of code of yours certainly has a lot to say about it. I think we can look at two main issues – firstly the use/misuse of macros and secondly the way enums are used. Oh, and use of global variables too makes it three.

OK, here's a statement – use of macros is bad! What do you have to say about that?

Well, when you say you can write a function once and then have many instances of it, I think you mean that you can write a function generator and then use it to generate a family of functions for you. But at what cost? Can you show me the code for your macro-generated functions? No? OK, well read up on your compiler settings. You can get it to spit out the C++ code which it sends to the actual compilation stage.

Right, what do you know? Yes you have a lot of gobbledygook in a big file (that's the **#include** statements bringing in the headers by the way), and somewhere near the bottom is your logic. This is a good exercise actually – when you write C++, you are actually writing in two languages – the preprocessor one, and then C++ itself. Seeing the output from the preprocessor is probably the best way to grasp that. Now, what can you tell me about your functions? Yes, there they are at the bottom. **LOG_ERROR()**, etc. By the way, pretty aggressive function names with all those capital letters. Yes, that might be a convention for the actual macros, but it doesn't have to output functions with all caps – look at your calling code, uggghhh!. And you'll have a job debugging this lot too, which is probably why you came to me.

I'd like to get rid of the macros before we go any further, then things might become clearer. Now, you want three similar functions without wanting to write the logic three times. That is very admirable, but let's achieve it using the language of the compiler, not the preprocessor. Now, we have a good start with the three functions **LogError()**, **LogWarning()**, **LogDebug()**. Yes, I've done a bit of renaming there. Now, what's the thing which varies between these? Yes, the 'level' of the information being logged.

We can implement that with a common function, can we not? Let's call it **Log()**. Now, it'll need to know which level we are talking about, and the message, so it's got two parameters – level and message. Exactly what the level is, we'll now talk about. Do you know, it'll look remarkably similar to your **Log** macro function, except that you didn't actually have a function, rather the function generator macro thingy.

What do you want to happen when these three functions are called? OK, output the message with appropriate level indicator, plus the additional logic that we only output the message at all if the global log level is set to include that.

Tell me about this level variable, then. Aha, it is a combination of all the options you wish to output. How have you encapsulated the concept of 'combination'? Bitwise **OR**/**AND**. Yes, that's a pretty standard way of dealing with options which are mutually independent. How does it work then?

The required values are **OR**'d together and the result contains them all together. Hmm, that's a bit vague, let's look in more detail. The clue is in what you just said, 'bitwise' **OR**. Each possible different value must be represented by a different bit, 001, 010, 100, etc. in binary.

I think the problem here is that you declared the variable as an **int**. However, we are not interested in level as a single integer, rather a combination of 'flags'. Let's make it something like a **BYTE**, or a **WORD**, which give the reader the anticipation that we are not interested in how many cups of tea the user wants, but something rather more geeky.

Let's look at the values you are using in more detail. So what are your values? Your enum, yes, we got to that eventually. Yes, but what are the actual values of your enumerations? You don't know. Ah, well to save you the round trip, let me tell you that if you don't specify values for enumerations, they start at 0 and automatically increase by 1 with each subsequent enumeration. So you have … 0, 1 and 2. Yes. Now, can we see the problem? How do you set the level to say you want to combine all three? 0 **OR** 1 **OR** 2, is? Yes, 3. But that is only two bits set, not three. You need to make sure that all of your options have an independent bit. Here's how:

```
Enum LogLevel { LogLevelError = 0x01,
   LogLevelWarning = 0x02, LogLevelDebug = 0x04 };
```

I put the numeric literals in hex, to give a clue to the reader that we are not interested in them as integers, but flags. Pity there is no binary literal syntax, eh? Oh, and I put **logLevel** at the start because the enum names leak out into the surrounding namespace and would clash with any others of the same name. [You can wrap them in a namespace for that purpose, that's a bit more advanced, though.]

Now, can you think what was happening to your errors? Look at your code in the macro generated function again. Yes, you are doing an **AND** with zero and getting ? Yes, zero back. Simple.

So, there's one last topic of concern to me – the dreaded global variable. It all hinges on level. First, that's something which might easily clash with

other variable names, but more importantly, it's a prime source of coupling in your code – testing this in a real-world project with lots of source files becomes onerous. I would recommend hiding this variable somewhere and having a function to set the log level, then you have the following functions in your library:

```
void SetLogOutputs( WORD levels );
void LogError( const std::string& message );
void LogWarning( const std::string& message );
void LogDebug( const std::string& message );
void Log( LogLevel level,
  const std::string& message );
```

Voila! Of course, there are the implementation details now, which I leave up to you, and we could spend a long time making this much more detailed and flexible – there are many logging frameworks out there, on which people have spent a lot of time, but that's for another day…

But, before we go, let's talk about something else. What does the above set of functions represent?

They are an interface. In this case a pretty C-style interface of a group of functions which might be in their own header, with their own source. With a little jiggery-pokery, we can put them as members of a C++ class. Give it the levels in the constructor, and we have a handy **`Log`** object we can pass around to the functions we call. That is what pattern exactly?

PFA, or PARAMATERISE FROM ABOVE! Now, in this model, you pass your log object around, and you can have as many as you like with different levels, different destinations, etc. You can even configure it from outside the program and have some settings module pick it up. Then you will be on the road to ultimate programming fulfillment.

Nevin Liber <nevin@eviloverlord.com>

The main issue is that enums are sequential numbers, not bit positions. The line **`if ( level & LEV )`** is the giveaway. To fix this:

```
enum Level
{
  Error = 1,
  Warn = Error << 1,
  Debug = Warn << 1
};
```

Now, there is a deeper issue, in that **`level`** is declared as an **`int`**, yet its possible values are permutations of the values in **`enum Level`**. Since these two things go hand in hand, they really should be represented by a single type. The obvious way is to make this a class; I'm going to explore the less obvious way by adding helpers to **`enum Level`**.

Most C++ operators (assignment being the notable exception) can be declared as non-member functions, which, among other things, allow us to add functionality to an enum. To support bitwise operations, I'm adding the following functions:

```
Level& operator&=(Level& l, int mask)
{
  l = static_cast<Level>(l & mask);
  return l;
}

Level operator|(Level l1, Level l2)
{
  return static_cast<Level>
    (l1 | static_cast<int>(l2));
}
```

All the messy casting is done in these functions, so that the code which uses this type doesn't have to worry about it.

I would also like a way to display all the bits that are set in a variable of type **`Level`**, so I add the following stream insertion operator:

```
std::ostream& operator<<
  (std::ostream& os, Level l)
{
  if (!l)
    return os << '0';
  char const* bar("");

  if (Error & l)
  {
    os << bar << "ERROR";
    l &= ~Error;
    bar = "|";
  }

  if (Warn & l)
  {
    os << bar << "WARN";
    l &= ~Warn;
    bar = "|";
  }

  if (Debug & l)
  {
    os << bar << "DEBUG";
    l &= ~Debug;
    bar = "|";
  }

  if (l)
    os << bar << static_cast<int>(l);
  return os;
}
```

This will not only output the bits as a textual representation, but if none of the bits are set, or extra bits are set, they will be displayed as a number as well.

Moving on to **`now()`**: the only use of this is to stream it out, so why pay the cost of a temporary string? For values to be streamed, a better method is to declare it as a struct or a class with a stream insertion operator declared as a friend, as in:

```
struct now
{
  explicit now(time_t timeNow = time(0))
  : timeNow(timeNow) {}
  friend std::ostream&
    operator<<(std::ostream& os, now const& t)
  {
    tm      local;
    char    buffer[sizeof("31 Jan 12:34:56")];

    strftime(buffer, sizeof(buffer),
      "%d %b %H:%M:%S",
      localtime_r(&t.timeNow, &local));

    return os << buffer;
  }
  time_t timeNow;
};
```

It is also more flexible, as it can produce formatted output for any time value. If someone does need it as a string, they can always use **`std::ostringstream`** or **`boost::lexical_cast`** to convert it.

Now to address the macro **`LOG(LEV, X)`**: given the drawbacks of macros (they manipulate syntax without regard to semantics), they generally

should only be used to express things that cannot be directly expressed in the language. Here it is used for two purposes:

1. To stream the textual representation of **LEV**.
2. To stream out any type **X**.

I have already addressed (1) with the stream insertion operator for **Level**; (2) can be directly represented in the language by using a template.

In addition, besides doing double work (and creating an extra string) by streaming into an ostringstream and using **printf** to actually display the string, there is a subtle flaw (which in some circumstances, can lead to an exploitable security hole) in the **printf** itself: if the output contains a **"%"**, it will be interpreted as formatting (taking whatever happens to be on the stack as the variable being formatted) instead of directly output. When using **printf**, always specify a formatting string (even if it is trivial) which is distinct from the data being formatted, as in **printf("%s", oss.str().c_str());**

Rewriting **LOG()**, we get:

```
template<typename T>
void Log(Level lev, T const& t, Level l)
{
  if (l & lev)
    std::cout << now() << " [" << lev
      << "] " << t << std::endl;
}
```

Note: I added the third parameter l so that this function is not reliant on **level** being a global variable.

Finally, I'll add a global level variable (this time of type **Level**, not **int**) for backwards compatibility, as well as rewriting the three logging functions as template functions:

```
Level level;

template<typename T>
void LogError(T const& t, Level l = level)
{ Log(Error, t, l); }

template<typename T>
void LogWarn(T const& t, Level l = level)
{ Log(Warn, t, l); }

template<typename T>
void LogDebug(T const& t, Level l = level)
{ Log(Debug, t, l); }
```

Making the syntax changes in **main()** (I only use all uppercase names for macros, not enum labels nor template functions):

```
int main()
{
  level = Debug | Warn | Error;

  LogDebug( "This is a test" );
  LogWarn( "This is a warning" );

  std::ostringstream oss;
  oss << "An example msg";

  LogError( "Problem: " + oss.str() );
}
```

One more thing: while the expression

```
"Problem: " + oss.str()
```

works correctly in that it returns a **std::string**, it is more likely that it worked accidentally than the author deliberately took advantage of the following free function being declared in **<string>**:

```
template<class charT, class traits,
  class Allocator>
basic_string<charT,traits,Allocator>
operator+(const charT* lhs, const
  basic_string<charT,traits, Allocator>& rhs);
```

It would be better to write this more explicitly, as in:

```
std::string("Problem: ") += oss.str()
```

It may seem contradictory because I'm taking advantage of the fact that **operator+=** is a member function of **std::basic_string**, not a free function (because in the non-member case, the temporary **std::string("Problem: ")** cannot be bound to a non-const reference, while it is perfectly okay to call member functions on a non-const reference). However, if it weren't, the worst that happens with my code is that it doesn't compile. In the case of the user code, if that overload for **operator+()** hadn't been provided, the result would be another subtle bug in the program.

So, one question to ask is: it took a bit of effort to add all this machinery to **enum Level**, **now()** and **Log()**; was it worth it? In practice, I've found that yes, it is. By having all that machinery in place, it is far easier for the people who use my code to use it correctly, not only in the regular case, but under higher pressure debugging circumstances as well.

Commentary

I think between them the entrants covered most of the points about this code, except for one. I was going to say except two, but Nevin's entry referred to the problem with **printf()** being passed a string, rather than a format string, as the first argument.

The remaining issue I would like to highlight is that there were several complaints about the problems caused by the use of pre-processor macros – and I agree with all their criticisms. However one advantage of the macro solution (and one reason why it is still used in many C++ projects) is that the compiler is able to make better optimisations. Consider this code:

```
LOG_DEBUG( anExpensiveCall() );
```

The code expands to

```
if ( level & DEBUG ) {
  std::ostringstream oss;
  oss << now() << " [" << "DEBUG" << "] "
  << anExpensiveCall()<< std::endl;
 printf( oss.str().c_str() );
}
```

The call to **anExpensiveCall()** appears inside the conditional and is therefore only made if the **DEBUG** logging level is enabled; whereas if we use a function call the compiler has to evaluate the arguments whether or not they are eventually used by the function.

Of course, this can be a problem if the calls have side-effects as the behaviour of the program can be affected by the logging level – just the problem Ken highlighted in the use of **assert**.

In many applications the unnecessary cost of evaluating the arguments needed for unwanted logging is high (in a previous project it totally dwarfed the cost of the application itself). In order to gain the same performance benefit without using macros you need to add a manual check before each call, something like this:

```
if (level & DEBUG )
   logDebug( anExpensiveCall() );
```

This is error prone – I've seen mistakes like:

```
if (level & INFO )
   logDebug( anExpensiveCall() );
```

and it also makes the logging more intrusive. So despite my general disapproval of macros there are some places where I consider their benefits outweigh their costs.

Listing 2

```
// ---- db.h -----
#include <string>
#include <vector>

namespace DB
{
  class Row
  {
  public:
    std::string toString() const;
    // ... other method elided
  };

  static struct exec_t
  {
  } execute;

  class SqlBase
  {
  public:
    void bind( int & );
    void bind( std::string & );
    // ... other methods elided
  };

  class Query : public SqlBase
  {
  public:
    Query( std::string const & query );
    std::vector<Row> operator<<( exec_t & );
  };

  class Insert : public SqlBase
  {
  public:
    Insert( std::string const & statement );
    void operator<<( exec_t & );
  };

  // Use template to retain type of 'stream'
  template <typename T, typename U>
  T & operator<<( T & stream, U & value )
  {
    stream.bind( value );
    return stream;
  }
}
```

Listing 2 (cont'd)

```
// ---- test.cpp -----
#include <fstream>
#include <iostream>
#include <string>
#include <vector>

#include "db.h"

int main()
{
  using namespace std;
  using namespace DB;

  int id;
  string name;
  string filename( "output.txt" );

  // populate id and name ...

  Insert( "insert into employee(name,id)"
    " values(%,%)" )
    << name << id << execute;

  vector<Row> result = Query( "select name "
    "from employee where id = %" )
    << id << execute;

  string firstrow = result[0].toString();
  ofstream( filename.c_str() ) << firstrow;
}
```

However, you must take care to avoid the sort of name clashes that the critique code had with **oss**, for example by using 'ugly' variable names.

The winner of CC 54

I was pleased to see Ken's entry this time round, where he was able to find the two biggest problems with the code by simply (!) running PC-Lint. While being able to read code and find faults is a fundamental skill, having the ability to use a tool to perform the same task enables much more reliable coverage of complete code bases. So this time I have decided to award the prize to PC-Lint for finding the problems; Ken gets the prize for using the right tool for the job!

Code Critique 55

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

I'm trying to write a C++ SQL framework that uses the **operator<<** idiom to add bind variables. I'm having problems getting the operators right – I've stripped it down to the following code that won't compile. Using MSVC I get a complaint about 'bind' on the last line of **main**; with g++ the **Insert** and **Query** examples won't compile either. Can someone help me sort this out?

You can also get the current problem from the accu-general mail list (next entry is posted around the last issue's deadline) or from the ACCU website (http://www.accu.org/journals/).

This particularly helps overseas members who typically get the magazine much later than members in the UK and Europe.

Bookcase

The latest roundup of book reviews.

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

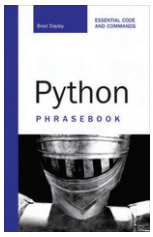
After you've made your choice, email me and if the book checks out on my database, you can have it. I will instruct you from there. Remember though, if the book review is such a stinker as to be awarded the most un-glamourous 'not recommended' rating, you are entitled to another book completely free.

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

Jez Higgins (jez@jezuk.co.uk)

Python

Python Phrasebook



By Brad Dayley, published by Sams, 2006, ISBN-13: 978-0672329104

Reviewed by: Gail Ollis

As a Python evangelist I've become rather resigned to people saying 'that's for web stuff, isn't it?', when I enthuse to them about how Python made programming fun again. It's like saying 'C++ – isn't that for banking applications?'. Sure, lots of people use it that way, but there are many more who don't. I was therefore rather disappointed to find half this book given over to the 'web stuff' – but maybe that's just because it seems to reinforce the misleading stereotype I've been struggling against.

As a consequence, it means that only the first half of the book is likely to be relevant unless you're doing web programming. That half is not bad as an aide-memoire because it covers bread-and-butter functions that any Python programmer will be using a lot – i.e. handling strings, files, data structures. The second half does the same thing for html, databases, xml and so on. It's day-to-day, basic instruction; the Python Phrasebook selects a few choice functions but doesn't tell you where to get the full story.

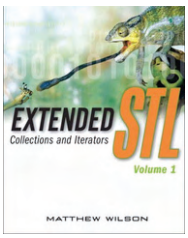
Far better to find out for yourself from the excellent online resources and make notes tailored to your own needs. This book feels a bit like someone else's notebook – someone not sufficiently well versed in the language to be the person you borrow notes from. Describing Python's class inheritance as 'similar to that in C' [sic] must surely be a typographical mix up, but there are other signs that this is a book by someone who finds a bit of Python useful rather than someone who understands it. The obsession with data types is confusing for beginners, while the misunderstanding of what 'dynamic' means – 'it's easy to get creative' – left me dumbfounded. At the ACCU conference static versus dynamic boat race in 2007, the author wouldn't have known which boat to join!

The 'phrases' themselves don't contain any such glaring errors, though a whole table of socket types is wrong in a classic cut-and-paste error; all the descriptions are in fact file modes copied from an earlier table. There's also a bizarre example of how to slice a dictionary; it's certainly not a common 'phrase' and I'm still trying to work out why anyone would possibly want to do it. But mostly, the errors are ones of omission: regular expressions, option parsing, iterators, generators and list comprehensions – all parts of an essential Python toolkit – are missing.

The claim on the back cover is that the Python Phrasebook 'lets you ditch all those bulky books', but with such omissions it can't achieve that end, leaving me wondering what the book is for. It's not designed to teach beginners, but holds little for the more experienced Python programmer. Perhaps the problem lies in the very concept of a phrasebook, a book of useful words and phrases in a foreign language. The linguistic version allows you to say things you don't understand and leaves you ill-equipped to deal with what happens next; good for comic effect but not for a professional programmer. As Monty Python's Hungarian phrasebook would say, my hovercraft is full of eels.

C++

Extended STL Volume 1



By Matthew Wilson, published by Addison-Wesley, 572 pages, ISBN: 978-0-321-30550-3

Reviewed by: Pete Goodliffe

This is an unusual book review for me. Normally, I digest a technical book in a matter of a week or two and then write a review. I don't like writing a review without having fully read the entire contents of a book; it's not a fair and representative review. Consequently this review is remarkably delayed as *Extended STL* has taken me literally months to read.

Why is this? The information in this book is dense and not easily digested. There's a lot of complex stuff in here that – especially in order to review fully – you really have to pay attention to. I'm not sure I can entirely blame this on the book, though. It's a natural consequence of the subject material.

Extended STL is the first volume in Wilson's projected two part series. This tome covers collections and iterators. That is, it describes how to create STL-compliant containers and iterators from scratch or how to wrap existing pieces of code with STL-interfacable proxies. This is a non-trivial area, with many subtle problems. The meat of the book is an in-depth description of problems and solutions in the implementation of STL-like code.

Volume 2 of the series will cover (amongst other things) STL-like functions, algorithms, adaptors and allocators. No doubt that, too, will be a dense book covering complex subject matter.

Extended STL is a fairly unusual book in the current marketplace, and so has little competition. There are many books on learning or using C++, on good C++ style, and on C++ programming idioms. There are many books describing particular C++ libraries. But there are few specifically about writing STL-like extensions, and interfacing existing code with the STL. So there's little competition for *Extended STL*. If you are doing this kind of work then the book looks like a sound investment.

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

This book is not an easy read. It took me an incredibly long time to complete; it's not easy to read in small chunks, or when your brain is full of other stuff. You really have to study a chapter in depth from start to finish to follow the flow.

Extended STL opens with a set of chapters on foundational concepts and terminology. These are a refresher in some important C++ STL-related techniques, and I suspect that the majority of readers will take away the most useful material from these chapters alone!

The chapters in the subsequent two parts (collections first, then iterators) usually cover a single example of the implementation of an STL-like component: explaining the reason for writing the component, the problems discovered during implementation, and the ultimate solution. An included CD contains all the numerous code examples in the book.

Sometimes this example-driven approach makes it hard to determine the most important information presented in each chapter. It also makes *Extended STL* less than ideal as a reference work in which to look up techniques.

The book contains a scattering of 'tips' in callout boxes. These appear to be a rather inconsistently applied narrative conceit. They should either have been scrapped or rationalised significantly.

As you read the text, it's clear that you are getting information from someone who really knows what he's talking about, and who has done a lot of this kind of legwork many times over. Wilson's writing style is clear, although sometimes I wonder whether the information could have been presented better in a different structure.

Extended STL is a testament to the incredible power and to the incredible complexity of C++. Many of C++'s detractors could cite it as a counter-example for use of the language!

I found very few technical problems or mistakes in the text. If I was being picky, I'd criticise the author's propensity for somewhat flowery language which will leave non-native (and some native) English speakers confused (or reaching for the dictionary).

In summary:

Pros

- Fairly unique coverage of this aspect of C++ coding
- Clear writing style
- The voice of an expert

Cons

- Dense information, often hard to digest
- Not ideal as a reference work

Recommended if you are writing STL-like C++ code, or want to interface legacy code with STL-style C++. Make sure you have plenty of time to sit down and digest the material.

PHP

Spring into PHP5

By Steven Holzner, published by Addison Wesley, ISBN: 0131498622

Reviewed by: Giuseppe Vacanti

PHP is one of the many languages that power web sites around the world, and probably one of the most popular. This book, published in 2005, describes version 5.0.0 of PHP (at the time of this review the production version is 5.2.6). Although PHP can also be used as a normal shell scripting language, the expectation, also in this book, is that PHP's home is inside a web page: most code fragments are in fact bracketed by some HTML mark up.

The book assumes very little previous programming knowledge. The first four chapters progress through what one might call the standard language constituents: variables and assignments, operators and flow control, strings and arrays, and functions. Every chapter is broken down into items, and each item is covered in one or two pages, with a sample HTML page to put it into context. If you have any knowledge of a C-type language, you will be zipping through these sections, as PHP's syntax has almost no surprises.

Chapters five and six cover how to handle HTML forms. The examples show you the HTML syntax for a certain form type (buttons, check boxes, text areas, etc.), and the PHP code needed to retrieve and validate the user's input. Again, the material is broken down into chunks that fit one or two pages, and the examples are clear.

One of the new features of PHP5 when it appeared was the ability to have classes and inheritance. This topic is covered in chapter seven, which for some strange reason also deals with file handling, although the latter could have been treated in a separate chapter. PHP5 has public, private, and protected inheritance, and classes are instantiated with new: the reader familiar with C++ will quickly come to grips with the syntax, which is clearly explained.

The examples dealing with classes are rather contrived, although they do the job of illustrating the syntax. Classes are defined, instantiated, and used in the same HTML page, which rather defeats the purpose of having classes in order to structure a large application. I shall return to this remark later. Another comment I have about this chapter is that exceptions are not mentioned. It could be that they were not present in the language in 2005, although having gone through the change log at php.net I have the impression that they were there from the start.

Chapter seven continues with how to deal with file access, while chapter eight describes how PHP5 can be used to access a database, with particular emphasis on MySQL. Again, a topic is enunciated, the required syntax is explained, an example follows, and the resulting web page is displayed. All in two pages, and rather effectively. The chapter also introduces the PEAR DB module, an abstraction layer that allows one to work with more database engines than MySQL alone.

The final chapter covers a variety of items, ranging from how to set and retrieve cookies, connect to an ftp server, and send email, to working with sessions. The book concludes with two appendices containing the language and function references.

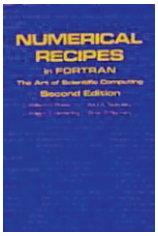
As I have hinted in a few places, the book is well written and well organized, with the exception of file access being grouped with object orientation. By the end of the book one gets a good impression of how PHP5 works and how it can be used; the chunk structure makes it easy to find what one needs when working in browse mode, for instance, when writing code.

I have however one criticism. By the end of the book it will become obvious, I hope, that mixing HTML and PHP5 snippets leads to untestable and poorly structured code. In the end, capturing the user's input and displaying the application's output must be kept separate from most of the application's logic. Of course, the ability to mix a programming language with HTML mark up is what makes the language attractive, and this is indeed a feature of many web application languages. However the areas where the two aspects come together must be kept to a minimum, and that is why in the PHP world templating systems like Smarty have been introduced. This book however completely ignores the topic, and this is not compatible with the statement that this book is for those who 'truly want to develop all the power of web applications'.

In summary, a good book, but not the only book on PHP5 you are likely to need if you wish to move beyond a web application that consists of a few HTML forms.

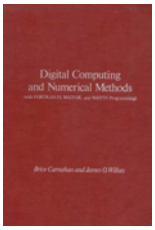
Fortran

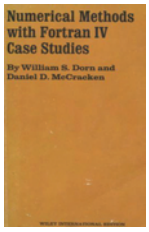
Numerical Recipes in FORTRAN: The Art of Scientific Computing, second edition



By William H. Press, Saul A. Teukolsky, William T. Vetterling, Brian P. Flannery, published by: Cambridge University Press 1992, 915 pages, ISBN: 052143064X

Digital Computing and Numerical Methods (with FORTRAN-IV, WATFOR, AND WATFIV Programming)





By Brice Carnahan and James O. Wilkes, published by John Wiley and Sons, 1973, 439 pages, ISBN: 0471135003

Numerical Methods with Fortran IV Case Studies

By William S. Dorn and Daniel D. McCracken, published by John Wiley and Sons, 1972, 403 pages, ISBN: 0471219193

Reviewed by: Colin Paul Gloster

This is a combined review of three books on numerical methods with FORTRAN.

N.B. The instalment of the famed 'Numerical Recipes' under review is not *Numerical Recipes in Fortran 90: The Art of Parallel Scientific Computing* dated 1996, but a version for an unofficial dialect based on FORTRAN-77. A variant of this book exists with '77' in the title (*Numerical Recipes in FORTRAN 77* – without a hyphen this time) with the exact same ISBN and supposedly also dated 1992.

The first chapter of *Numerical Methods with Fortran IV Case Studies* begins with good examples of how difficult it is to capture the intended semantics of an approximate mathematical solution being 'near' an exact solution. Of the three books reviewed here, this is the best one for maintaining a comprehensible manner when describing the mathematics involved. The next best in this regard is *Digital Computing and Numerical Methods*. The assumed knowledge of the readers of *Numerical Recipes in FORTRAN* is not uniform, but in most cases reading its chapters without an introductory book from my teenage years would leave me at a disadvantage. Fourier and wavelet transformations, statistics and topics which are not the main focus of the book are all explained under the assumption that their basic fundamental notions are understood.

In some cases the assumptions regarding the readers of *Numerical Recipes* are not uniform because different topics are being presented, but sometimes it seems as if the coauthors had not decided exactly who they were writing for. For example, bins are mentioned independently in Sections 7.8, 13.4 and 14.3 without referencing the other sections and without being listed in the table of contents nor the index. It is assumed in Section 7.8 that the reader knows what a bin is, unlike in the other sections. Similarly but to a less severe extent, Bayes's Theorem is printed a few pages later than where familiarity with it is actually needed.

Though fundamental concepts are explained in the other two books in a manner in which almost anyone could understand, those books are too short and describe too few methods. The numerical methods described in *Digital Computing* were restricted to pages 255-439. A vast amount of space is used in the rest of the book to describe number bases and punched cards. This is not to say that all worthwhile methods appear in *Numerical Recipes*, but it does contain many more. Some examples of topics which are not covered in any of the three books under review are concurrency, the Doolittle method and Adams-Moulton methods. The quotation from Byte magazine in the blurb of *Numerical Recipes* stating that the book is 'remarkably complete' is therefore untruthful. A quotation in the blurb from the *Journal of Nuclear Medicine* is actually contradicted by the authors on the first page of Chapter 6: Special Functions, who 'do not certify that the routines are perfect'.

Furthermore, Cambridge University Press misleadingly presented two quotations about *Numerical Recipes* out of context. Elizabeth Greenwell Yanik's remark in SIAM Review:

> The routines are prefaced with lucid, self-contained explanations

was quoted without the third next sentence:

> However, there are a number of advanced topics that require a substantial amount of collateral reading from the cited references in order to understand the algorithm.

The praise in the blurb attributed to Byte has been contradicted in a review published by the *American Journal of Physics*. Interestingly, the review in the *American Journal of Physics* was also quoted in the blurb, though its contradiction was omitted!

Much worse, somehow the only part of Jesse L. Barlow's review [1] of *Numerical Recipes* which is quoted is the first two sentences of the following paragraph:

> The book's virtues are that it lists important topics from numerical analysis that may be of interest to scientists and engineers. It gives a summary of the philosophy behind each of the methods discussed and a bibliography so that one can find out more. Unfortunately, the bibliography is somewhat dated. The computer programs in this book should be avoided. The book might be used as a starting point, but the reader should always look up a topic in another source.

Why was Barlow so negative? The best way to check is to read Barlow's review in full. Anyone who might rely on *Numerical Recipes* should be forced to read Barlow's review, along with Jim Law's review in *ACM SIGSOFT* [2] and part of *Scientific Computing FAQ: Books, With and Without Software*, for NA, by Steve J. Sullivan [3] . Though the details of Barlow's review are debatable, Barlow has the remarkably important status of being one of only two reviewers (whose reviews I have read) in which it is pointed out that the coauthors are not wise concerning mathematics (I have read 64 reviews of installments of the *Numerical Recipes* series). The other reviewer who decried the mathematics of *Numerical Recipes* is Frederick N. Fritsch in the January 1988 issue of *Mathematics of Computation*. For example, on page 53 they show that they do not understand that there is no such thing as 'the condition number' of a matrix. Condition numbers are like norms in the sense that many different types of incompatible definitions of condition numbers exist, instead of merely 'the' condition number. Condition numbers do not appear in any of the other books under review.

I believe that the idea of aliens flying UFOs is symbolic of a religion for people who have no religion. Considering the reputation of *Numerical Recipes* and that over twenty of its reviews were completely lacking any mention of inadequacies, I believe that I may have found another new religion of the twentieth century. It may be understandable that incidental programmers and incidental mathematicians such as physicists and biologists might not have noticed various inadequacies in *Numerical Recipes*, but the reviewers for *Byte* magazine and *SIAM Review* (and many of the reviews published by the Association for Computing Machinery: they were not all accurate) should not be so easily excused.

A disadvantage of *Numerical Recipes* is that it contains source code. Without this source code, a reader would be forced to devise subroutines which might be more appropriate than the examples in the book. However, many of the readers actually prefer that it comes with source code (apparently oblivious to the coauthors' warnings to not use them). This harmful misplaced belief of what they need (instead of what they truly need) has been expressed in a number of reviews of various installments of the *Numerical Recipes* series, and taken to its extreme in the following quotation from a review of *Example Book (Numerical Recipes) FORTRAN* and *Example Book (Numerical Recipes) PASCAL*:

> The example books contain programs that allow the routines given in *Numerical Recipes* to be embedded in a useful program environment, i.e. as complete user software packages not unlike the solution packages that are available for the Personal Computer market. It is this approach that has been missing for the field of scientific computing, and these example books fulfill a real need.

The other two books under review come with even worse code (from a software engineering perspective), partially because of the use of Fortran IV instead of Fortran 77. However, much of the code in *Numerical Methods* is worse than the best possible in Fortran IV. For example, the first subroutine call in that book is on page 173, yet subroutines had supposedly been added in Fortran II in 1958.

Much as incidental programmers do in any language in 2008, *Digital Computing and Numerical Methods* contains incohesive data initialization statements such as:

```
DATA X, Y(6), INT, FIRST, LAST,
  SWITCH/ -6.7, 1.93E-6, 49,
  1 'JOHN', 'DOE', .TRUE./
```

which admittedly is milder than what exists outside of the books. The *Numerical Recipes* review by C. R. Jenkins in the February 1987 issue of *Observatory* states:

> My favourite NAG [Numerical Algorithms Group, a library vendor] subroutine, for example, has no fewer than 25 parameters.

Do not worry! Nothing in any of the three books under review is that bad.

Not every piece of code in *Digital Computing* is bad however. For example:

```
ALOG(ABS(TAN(ALPHA)))
```

is a violation of the Law of Demeter but it is sensible.

It would be possible to rewrite much of the software in *Numerical Recipes* in order to use fewer statements for the same outputs. For example, the lengths of **sprspm** and **sprstm** are nearly 56 lines each and they differ by approximately only twenty lines. However, much of the software in the book exhibits less dramatic, debatably poor, software engineering. If you have prospective clients or colleagues who are incidental programmers who learnt from this book, then it may be worthwhile choosing some pages containing lines like:

```
sigl=max(TINY,
   fmaxl(j)-fminl(j))**0.6666)
sigr=max(TINY,
   fmaxr(j)-fminr(j))**0.6666)
```

and showing them how you would replace those lines. If they accept the improvements, then you may all win. If you do not convince them, it may be an early warning that you should not become involved. If you do not realize what might be poor software engineering in those lines, then avoid this book because it would not improve your programming skills!

A sample of the new 2007 C++ edition of *Numerical Recipes* [4], which is supposedly object-oriented and efficient, contains:

```
Int i,ii,j,jj,np,schg=0,wind=0;
Initialize sign change and
winding number.
...
p0 = vt[0].x[0]-vt[np-1].x[0];
p1 = vt[0].x[1]-vt[np-1].x[1];
...
for (i=0,ii=1; i<np; i++,ii++) {
Loop over edges.
...
    d0 = vt[ii].x[0]-vt[i].x[0];
    d1 = vt[ii].x[1]-vt[i].x[1];
...
```

Note the capital **I** in class **Int** warns that it is not an int! One of the problems encouraged by the above piece of C++ code is postincrementing of objects (postincrementing of many kinds of objects in C++ can be slower than preincrementing). Other issues with the scheme chosen to control the **for** loop could also be remarked upon.

We can also see that copying and pasting code is a dubious manner of reusing code which they still have not abandoned. Jim Law's review [2] of the 2002 (not 2007) C++ version is accurate (though slightly harsh). He deplored the incoherent intended readership ('experts who would deem the book to be insufficient or beginners who would not learn enough from it'); and 'simple rewrite of the C programs in C++':

> There is no attempt to understand any relative strengths or weaknesses of C++ and object-oriented programming, or to accommodate C++ users.

It does not seem that any version of any edition of this work is excellent in these respects. C++ may have advantages over Fortran 77, but the coauthors should have spent time improving their programming skills before transferring their programs to another language.

However, *Numerical Recipes* does not provide convincing evidence that the coauthors even learnt Fortran. I have never read the Fortran 77 standard, so perhaps every Fortran 77 book, compiler and editor I have ever used contains mistakes, but according to all these sources a continuation line can not have an asterisk in the first column (yet an asterisk in the sixth column would be acceptable). However, a supposed continuation line in *Numerical Recipes* is denoted by an asterisk in the first column, thereby making it a comment, thereby making the software uncompilable.

Documentation is not the best in *Numerical Recipes*. For example, **SUBROUTINE bandec** contains:

```
if(mm.gt.mp.or.m1.gt.mpl.or.n.gt.
np) pause 'bad args in bandec'
```

which is the only explanation of the meaning of the argument **mpl**.

In *Pascal for Science and Engineering* (which I reviewed in the November 2008 issue of C Vu), and *Numerical Methods with Fortran IV Case Studies*, it was reassured that numerical integration is relatively easy. This seemed to be supported in one of the chapters on integration in *Numerical Recipes in FORTRAN* but was contradicted in another of its integral chapters with a warning of 'many different possible pitfalls'.

The Gauss-Seidel and Jacobi methods were briefly mentioned in *Numerical Recipes* and dismissed. These methods are not ideal, but they were treated harshly in *Numerical Recipes* and they are very useful for explaining tradeoffs between concurrency and lack of concurrency. Books in which these methods and concurrency are discussed include *Applied Numerical Analysis* by Curtis F. Gerald and Patrick O. Wheatley, *Numerical analysis for applied science* by Myron B. Allen III and Eli L. Isaacson, and *Teach Yourself Algorithms* by Anthony Ralston and Hugh Neill (which is in the same series as *Teach Yourself Flower Arranging*).

Horner's method was shown, explained and named in *Digital Computing* and in *Numerical Methods*. Horner's method is something which a good compiler should implement automatically. In *Numerical Recipes* it must be assumed readers already know it, as it is not explained nor named therein. The more obvious notation that can be used instead of Horner's method is not efficient, and it was joked in *Numerical Recipes* that people who use this should be 'executed', yet in their **SUBROUTINE trncst**, in a different chapter, they have used an obvious algorithm (i.e. calculating distances by square roots) to choose a short route. They should appreciate that this is not efficient if the values of the distances are not actually important, but merely their relationships to each other. (N.B. An efficient alternative is shown in *3D Game Engine Design: A Practical Approach to Real-Time Computer Graphics* by David H. Eberly. If the values of the distances are important, then an efficient technique in one of André LaMothe's chapters in *Tricks of the Game Programming Gurus* could be used instead).

I rarely see Fortran's **COMPLEX** mode ('mode' is Fortran jargon for 'type'), in books, but I was once asked by a Fortran programmer whether another language as opposed to a library has a **COMPLEX** type. In *Numerical Recipes* it is claimed that compilers' implementations of **COMPLEX** are buggy and/or inefficient.

Some other points of note:

The software in *Digital Computing* is not rigorous, and in more than one chapter, software in *Numerical Recipes* does not guard against misuse, unlike the more defensive programming seen in *Numerical Methods*.

There are conflicting warnings regarding double precision in *Numerical Recipes* on pages 731 and 882. In *Numerical Methods*, sparse matrices were not covered sufficiently. In *Digital Computing*, it is claimed that functions can be used with the complete confidence that they produce the necessary results (you should not be so credulous!).

Digital Computing does not always teach the simplest method before a more advanced method. Some parts of this book are good but short. Too many other parts are bad.

A surprisingly (worryingly) small amount of complexity theory (big-O notation), is present in these books, with none whatsoever in *Digital Computing*.

Numerical Methods has a much better treatment of errors than the other books under review. It contains good examples showing that increasing the quantity of terms or digits can actually result in less accurate results. A perfect book on numerical methods would merge the *Numerical Methods* treatment of errors with the treatment in *Applied Numerical Analysis* by Curtis F. Gerald and Patrick O. Wheatley. However, *Numerical Methods* dismisses worst case bounds on errors as being excessively conservative. I disapprove of this attitude, which is also found in *Digital Computing*.

As a point of interest, in a Sun Microsystems article published in *Integrated System Design* magazine in 1996, it was claimed that to 'construct by correction' was a major part of a

View From The Chair

Jez Higgins
chair@accu.org

In my household the New Year marks the start of the annual negotiating period before the ACCU conference. If, the reasoning goes, I'm off on a jolly for four days, what does the rest of the family get in return. My arguments that going to the conference is work, helping to keep the roof over the head, food in the dogs' bowls, and so on falls on deaf ears. "Yeah Dad, you can stay up as late as you like when you're away", was apparently so crushing that no response could counter it, even though as a grown up I can, unlike 8 year old boys, go to bed when I like whenever I like. Going to the conference is work - I regard it as part of my informal continuing professional development - and it can be hard work too. Four days is a long time to be continually focusing, learning, filtering, talking. I generally come away feeling completely exhausted, even overwhelmed, taking weeks, months or even years to chew over and think through what I've seen and heard. But, by heck, isn't it fun?

This year's conference runs from the 22 to 25 April, which is a little later than last year, at the Barcelo Hotel in Oxford. It features keynote presentations by Robert "Uncle Bob" Martin, Frank Buschmann, Baroness Susan Greenfield, and our own Allan Kelly. The conference committee are confirming the programme as I write, and it will, I suspect, be up at http://accu.org/conference by the time you read this.

The ACCU's annual general meeting also takes place during the conference. One important item on the agenda each year is the election and reelection of the committee and the officers. While you can get involved in the running of the organisation at anytime, simply by offering, the AGM is an obvious start point. If there's something you feel the ACCU could or should be doing, or something it's doing now that could be better, then consider standing for the committee. You don't have to come brimming with revolutionary fervour, you might simply be willing to help and that's just as important. Being on the committee needn't represent a huge time investment, but as a volunteer run organisation everybody's contribution can help to make a difference. Do drop me a line if you'd like to discuss an idea you have, or to find out more.

Membership

Mick Brooks
accumembership@accu.org

Contact the memberhip secretary with any questions about membership or journal delivery.

Bookcase (continued)

new methodology devised for designing UltraSPARC-1 processors. However, the technique was not new because it had been described, without the buzzwords, in *Digital Computing*, 1973 as:

> Virtually every engineering design evolves in an iterative or trial-and-error fashion.

In summary, for the purpose of numerical methods, *Numerical Methods with Fortran IV Case Studies* is highly recommended because of its excellent treatment of errors and also because it is easy to understand. Afterwards, a book on other numerical methods should be read (ideally not one of the other two books which are under review). The other two books reviewed here are so bad as to be merely recommended with reservations, although the treatment of least squares fitting in *Numerical Recipes* is the best of all books I have ever read.

For the purpose of checking programming skill levels of incidental Fortran programmers, any of these or a number of other books could be used.

I have used some of the excellent *Algorithms for Programmers* [5] for extremely efficient numerics not dealt with in any of the three numerical Fortran books under review. I have not yet read any of the parts of *Algorithms for Programmers* which have exactly corresponding sections in any of the Fortran books, but judging from what I have read, it may be worthwhile considering this instead of buying an expensive reviewed book.

I am grateful to the American Library Association, the American Statistical Association and the Canadian Association of Physicists for providing me with some of the reviews mentioned herein. I am grateful to Peter Friis and Stuart Golodetz for helping me to obtain other reviews. I unfortunately have not obtained all of the reviews of *Numerical Recipes* which had appeared in *University Computing*, *Dr. Dobb's Journal*, *Acta Applicandae Mathematicae*, *Binary*, *Boston Computer Society Newsletter*, *Bulletin of Mathematical Biology*, *Mathematics of Computing*, *SIAM News*, *The Mathematical Gazette* and *ZDM* (Zentralblatt für Didaktik der Mathematik/The International Journal on Mathematics Education). I especially still want to read those missing reviews, so I would be grateful if someone could help me to obtain them. I have asked the publisher and coauthors of *Numerical Recipes* to identify reviews in the blurb which I had failed to track down. At the time of writing one coauthor has provided me with a helpful list of some of the missing reviews, and the publisher has responded but not yet provided tangible help, though I had asked more than five weeks before submitting this review to the ACCU.

If you would like me to identify reviews discussed here please contact me for more information. You can try to email Colin_Paul_Gloster@ACM.org or fax +351 239829158

References

1. Computing Reviews, review number CR114306. Association for Computing Machinery
2. Software Engineering Notes, by Jim Law, ACM SIGSOFT March 2003
3. Scientific Computing FAQ: Books, With and Without Software, for NA, by Steve J. Sullivan http://www.mathcom.com/corpdir/techinfo.mdir/q165.html
4. C++ edition of Numerical Recipes http://www.nr.com/nr3sample.pdf
5. Algorithms for Programmers by Jörg Arndt http:www.jjj.de/fxt/

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