contents

credits & contacts

Overload Editor:
John Merrells
overload@accu.org

Contributing Editor:
Alan Griffiths
alan@octopull.demon.co.uk

Readers:

Ian Bruntlett
IanBruntlett@antiqs.uklinux.net

Phil Bass
phil@stoneymanor.demon.co.uk

Mark Radford
twonine@twonine.demon.co.uk

Thaddaeus Frogley
t.frogley@ntlworld.com

Richard Blundell
richard.blundell@metapraxis.com

Advertising:
Chris Lowe,
Pete Goodliffe
ads@accu.org

Overload is a publication of the ACCU. For details of the ACCU and other ACCU publications and activities, see the ACCU website.

ACCU Website:
http://www.accu.org/

Information and Membership:
Join on the website or contact

David Hodge
membership@accu.org

ACCU Chair:
Ewan Milne
chair@accu.org

Editorial – The state of software development

A few weeks ago I returned from holiday to find a lot of postings on the `accu-general` mailing list that had been initiated by my last editorial. The thing that started them was a posting by Allan Kelly (which also appears as a "letter to the editor(s)" in this issue). The thing that struck me about the discussion was that it generalised rapidly from the evolution of Overload to the evolution of software development as a profession.

Software development as a profession

Is software development a profession? Clearly, many contributors to the mailing list feel that it should be – they bemoan the lack of accreditation that is available to architects, doctors, accountants, and so on. Some of them even proposed that the ACCU might fulfil this role! (In the UK there is an organisation better constituted to do this – the BCS – but it lacks the recognition and authority that would be needed.)

How does a profession come to be? It requires several things: for the professional it requires there to be a strong incentive to be recognised as a professional (this could be legal, financial, or access to market). For the client, a strong incentive to employ a professional (this could be legal, guarantee of quality, or access to market). Is this what we see in software development?

At the weekend I was talking to a friend of mine from university, he works in software development and is a qualified project manager. To me this was a novelty – most of the project managers I've worked with have no qualification and, indeed, don't seem to have ever opened a book on the subject. (Mostly they have attained the role because that is what you do after enough years as an analyst-programmer, business analyst, or whatever.) It seems that his qualification (Prince 2) has some market value – there are organisations that require it, but hasn't been enough to keep him in employment recently. It is also hard to assess for those not familiar with it: it relates to a particular method which may or may not be suitable for a specific project.

The situation is worse for developers – what recognition do developers get for having enough interest in what they are doing to open a book? There are qualifications in particular technologies – but these are often short-lived and don't always reflect a developer's primary skill: that of solving problems. This isn't just a problem for the developer, but also for the prospective employer: how are they to identify the useful candidates?

This last question has vexed me somewhat over the last couple of weeks – I've been doing technical assessments of candidates for a client. Before I see candidates their CVs have been reviewed and the best selected, followed by eliminating those that get poor scores on BrainBench – but despite this selection process the majority are clearly unsuitable. (For example, when presented with a class hierarchy that makes use of the implementation of "cat" by specialising it to produce "dog" they approve of this approach to the reuse of code.)

But it is not typical of the job market for an employer to care about such fine distinctions between developers – despite my misgivings about their abilities almost all of these developers are in work and have succeeded in a series of jobs. Typically they assume they have done well enough on the test – displaying a confidence in their abilities, which appears to reflect that they are able to deliver what their employers expect of them.

To me it is clear that the majority of developers and the majority of their employers neither recognise nor care about the qualities that have been represented as "professional" in the ACCU. And yet these businesses are commercially viable – they are not being out-competed by a few more discerning organisations. It would appear that, for the business that employs them, there is no commercial value to professionalism in software development staff.

To me this is something of a paradox: I have repeatedly demonstrated ways to reduce the cost and timescale of software development. The size of the savings can be considerable – in one organisation a cost overrun of 120% was typical (and a significant part of the turnover), reducing this to 15% was not recognised and the "traditional" approach has been reinstated. (The other difference, which affected developers more than business costs, was a similar reduction in unpaid overtime.) Professionalism in software development does save time and money – it can be demonstrated; it is repeatedly reported in books and journals; and, it is rarely practised.

Where does this leave software development? Clearly, it doesn't meet the conditions I outlined above as being necessary for development as a profession. It isn't a profession now and it won't be in the foreseeable future.

Does this mean that there is no point in being "professional" in the way we approach it? I hope not: the habit of doing things right is a part of who I am – and I have enough experience of the pain that doing things wrong engenders to want to avoid it. I know that a lot of you feel likewise – the ACCU seems to have attracted developers who care about what they do. But, despite the ambitions of some, it isn't a profession.

Software development as a craft

Is software development a craft? In many ways this seems a much more appropriate analogy. The degree to which it relies on the practice of good judgement, and the extent to which this relies on experience is suggestive of this. There are often different "schools" within a craft and the variations between different development methods and tools fits well, as does the importance of the teachings of various masters. (Clearly the masters are those who are sought out for this purpose, not those who appoint themselves.)

The existence of several schools of software development is clear: there are "heavyweight" methods (like RUP); there are lightweight methods (like Crystal); there are weakly typed languages (like SmallTalk); and, there are strongly typed languages (like Java). Some proponents of each of these tend to believe that they alone have seen the one true way – while more measured observation will show that each has both successes and failures. (The existence of several schools isn't incompatible with being a profession – but it does make agreeing the criteria for any professional qualification more difficult.)

What is missing from "software development as a craft" is a recognised system of apprenticeships, journeymen and masters. Very few organisations consider that "I studied with Angelica Langer" is a better qualification than "I glanced at a book once" – and neither, judging by the CVs I see, do developers. (In case you are wondering I chose the example because there is at least one organisation that looks for this specific qualification – they know who they are.)

Does this mean that there is no point in seeking out the masters to learn from them? Again, I hope not: there is far too much to learn for trial and error to be an effective approach (especially when it is often hard to be sure which factors affect the result). The ACCU contributes here by providing several forums in which the masters can be sought: mailing lists, journals and conferences.

Software development as a science

Is software development a science? It is tempting to think that the existence of "computer science" courses provides some basis for this – but the true sciences (mathematics, physics, etc.) don't have "science" in their name. And a closer examination of "Computer Science" shows it to be mostly technology (there is some applied science – mostly mathematics).

The essence of science is the ability to devise experiments to test theories (at least that is how Popper and I see it). And in software development this is singularly hard to do. There is too much uncontrolled variability – and it isn't clear which variables are important. There are people trying to make a science of it: Alastair Cockburn collects anecdotal evidence about projects and tries to identify common factors – but this work is still at a stage where there is a lot of "interpretation" which may not sufficiently repeatable to support experimentation.

One day there may be a science of software development, but as far as I can tell it is a long way off.

Software development as a team activity

One classification I had not considered until recently is the idea that the important classification of software development is that it is a form of teamworking. This certainly fits with my often repeated plea for communication over documentation. And (as I discovered recently when reading an unpublished thesis on the subject – I hope to bring you some of this material in the future) there are various established classifications of team behaviour and development that are very useful in assessing development projects.

Oddly, this had not occurred to me as a useful classification despite its use in one of the first discussions I ever read on development processes (The Mythical Man Month). This uses analogies to teams in several different fields of endeavour to establish the roles and interactions that have proven successful.

On reflection though I think this is the right answer: looking back over various projects I find a strong correlation between the quality of the team (rather than of the individuals) and the results of the project. (The only exceptions I can think of are where a good team failed – and I would ascribe the failure to external factors.)

Software development and Overload

The remit of the ACCU has drifted over the years, it started with a very narrow focus on C as the "C Users Group (UK)". It has expanded into other languages (Overload was originally the journal of the Borland C++ User Group) and other areas of development.

Overload has always been more focussed on the "front line" rather than on the "supply lines" – which has its risks as well as its benefits. There is clearly interest in software development as a professional activity, as a craft, as a team activity and as an organisational function. These interests are not always compatible, but we can try to cover them all.

What appears in Overload is very dependent upon what is submitted for publication, and this covers an increasing range of topics. But there is a commitment from the team of readers to ensure that every article is as good as we can make it. You can do your part by submitting great material – or just by encouraging the authors of things you like.

It is up to you.

Alan Griffiths
alan@octopull.demon.co.uk

Copy Deadlines

All articles intended for publication in *Overload 58* should be submitted to the editor by November 1st, and for *Overload 59* by January 1st 2004.

Letters to the Editor(s)

The dbg library

Since my article on debugging and the dbg library was published in Overload 46, I've had a large number of readers commenting on how useful the library is, and even more positive feedback from programmers who've found it "out on the net".

Overload readers may be interested to know that the official homepage has now moved to `http://dbg.sf.net/`. The library has had several updates since the article was published, and is still under active development.

Pete Goodliffe
`pete@cthree.org`

The ACCU is a three-legged stool

Dear Editor(s),

First can I thank you for the Overload 56 editorial, it was well written, well timed and highly relevant. Maybe I shouldn't comment on it because I am one of those who has been pushing Overload in a non-programming direction, but maybe that means I also have something relevant to say.

I've been conscious for a while that some of my pieces are drifting away from core remit of "programming". However, I do get the occasional e-mail from people, and I did speak to a few people at this year's conference and nobody has said "Off topic! Stop!", in fact, my opinion is that people like this stuff.

I also recall that several of the non-programming sessions at the conference were very well attended, e.g. Hubert Matthews, "The Extreme Hour" and Alan Griffith's own "Too Agile?" sessions.

In my opinion what is happening is that ACCU is maturing, from a troublesome teenager into an promising 20-something. The organisation and its membership are no longer entry-level programmers, they have senior, even managerial positions. The choice ACCU must make is whether we say "Sorry guys, we are strictly programmers, managers not wanted" or whether we try to accommodate all.

To my mind the answer has to be the latter. If the organisation does not stretch and grow it will forever be a temporary home for a few programmers. Sure, it is fun to complain about managers on `accu-general` but we either spend our life complaining or we take responsibility for doing something about it. We have to engage with management, and academics, another of our pet-hates.

The differentiation between C Vu and Overload is clearer this month than it has been for a long time. C Vu is the journal of the ACCU, a dialogue between the officers and the members, it is also the home of new talent – both in writing and software development. Overload is maturing into a serious software development journal – it needs to broaden its remit and its readership.

You'll notice that I have deliberately blurred the distinction between the journals and the organisation. For me, ACCU is a three-legged stool: journals, conference and mailing list. These support the membership and help them grow. As the members grow so too does the organisation, mentored developers has appeared and pub meets are becoming more regular. I'm convinced some cities in the UK, specifically London, could copy Reg Charney's excellent monthly meetings in San Jose.

There is another reason why this debate is about more than the journals. Each year at conference we are reminded of the need to "stay under the VAT limit." (For those not in the UK, this is the level where an organisation must start to charge and pay sales tax.) As our membership continues to grow we will breach this limit, clever accounting can only delay this so long. When this happens the ACCU will have to consider its options carefully, it is better this happens before the event than after.

ACCU, and specifically the journals, need to grow, they need to stretch and raise their sights. I'm reminded of a quote I once heard from Christopher Alexander, I don't remember it verbatim but it was along the lines of "I'm frequently disappointed that young architects don't aim higher". I think it is time for the ACCU to aim higher.

Allan Kelly
`allan@allankelly.net`

Stream-based parsing in C++

Fundamentally, this paper describes a technique for writing recursive-descent parsers in C++ using operator overloading so that the driving logic of the parser is written in a way that looks like the parsed grammar rules. As presented, I think the technique is not usable, but it is not completely flawed and can be made usable.

My first comment is on usefulness. The author states "while C++ is certainly capable of handling recursion, the language is not really capable of handling recursive definitions such as the grammar in its present state". I'm not so sure I understand that sentence correctly as there is no explanation of what is lacking to handle "recursive definitions such as the grammar in its present state". I think that what is hinted at is that applying the given technique to a (directly or not) recursive grammar would produce code with infinite recursion. While some use of recursion can be removed by using closures (which the technique is able to handle by special casing), not all can and I don't think parsing techniques unable to handle recursion in a grammar are particularly useful. I don't know how this problem is removed in functional programming languages (Haskell could perhaps handle it using lazy evaluation, but not Common Lisp). Related to this, the author presents a second grammar as a rewriting of the original one with recursion removed. There is no recursion in the second grammar, but the parsed language is quite different to the one of the first grammar (and I don't think rewriting could remove recursion for that grammar while still parsing the same language).

My second comment is on the error handling, reporting and recovery. There is no explanation of the way errors in the parsed input are supposed to be reported, and as the technique uses backtracking, the naive way is not valid. The code presented just doesn't consume anything on the input in the presence of an error, and nothing is present in the output to indicate that an error occurred. There is also no error recovery strategy, but that's a minor point as I don't know a general parsing technique using backtracking with an error recovery strategy.

[concluded at foot of next page]

Is IT worth it?

by Allan Kelly

If you're a dedicated software developer you probably read a lot. I always think it pays to read widely, get outside the software field and expose yourself to some new ideas. While for some developers outside the field means swapping Sutter's *Exceptional C++* for Hofstadter's *Gödel, Escher and Bach* some of use do venture as far as *The Economist*, and maybe the odd business book. After all, IT is itself a massive business and, if we believe the hype, can benefit real businesses.

If you have ventured into the business press lately you will have found business writers and mangers questioning the true value of IT. This has been going on since the Y2K bug, dot-com bust and telecoms crash but has picked up momentum in the last few months thanks to an article entitled *IT Doesn't Matter* in the Harvard Business Review.

Now the HBR (as aficionados call it) is a rather august journal that sees itself at the heart of the business debate. Its detractors may suggest that many more copies are bought than are actually read but it still has the capacity to ignite a debate, which is just what *IT Doesn't Matter* did.

What did he say?

Nicholas G. Carr is a business writer and consultant, and a past editor of HBR. So when he writes about IT he is writing from a business perspective. Before you turn the page, remember that it is business that pays us. Unless you flip burgers by day and write Linux by night the chances are that at some point you need to consider IT from a business perspective too.

Carr's argument runs something like this. IT is really a form of infrastructure, like the telegraph or railways. During the last 20 years or so we have been building this infrastructure, not just the internet but putting PCs on people's desks, writing word processors, CRM packages, etc. During this time there has been an advantage to getting there first. That is, the company that is first to install word processors would see a business advantage, a *competitive advantage* to use business speak.

However, says Carr, that time is past. The infrastructure is now built. There is no point in you upgrading your secretaries to Pentium 19s because (a) it will cost you, and (b) it doesn't offer that much advantage over your competitor where they all have Pentium 18s.

In the process IT has become a commodity. You can buy 500g of IT like you buy 500g of cheese at the local deli. Look at the net services being promoted by Microsoft, IBM and others to see this in action. "Computing on demand" – you can switch it on and off like electricity, 100Mips of computing power may come from a Linux server farm in Aberdeen or a mainframe in Mumbia.

Now, since IT is a commodity there is no differentiating factor, there is nothing unique, therefore it cannot be the basis of competitive advantage. Even though IT may be essential to our operations it is not something we can use to gain an edge on our competitor. If I need 100Mips of power to run my business then so be it, I can buy it and so can my competitor, therefore it is not something that gives me an advantage. The only difference is how much each of us pay our suppliers for it, I need to drive a harder bargain with IBM than my competitor drove with EDS.

Even if you come up with some revolutionary new idea for using IT you still can't create an advantage because it is easy for anyone to copy.

Carr makes a good argument. He shows how IT has followed the same path of previous infrastructure technologies, complete with investor bubble and bust.

In conclusion he suggests we need to move "from offense to defense". Rather than spending vast sums on IT projects we need to look to manage the risks associated with IT – such as network

[continued on next page]

My next comment is on performance. The author states that the code presented is not optimised for speed of execution, nor for space, but is not slow. How do you qualify a parser with an exponential behaviour in terms of input length? Indeed, alternation is handled by backtracking and using backtracking without early termination due to error detection can lead to exponential time behaviour even in simple grammars which are parsable using LL(1) parser. A comparatively minor inefficiency is that the remaining input is copied around and in most grammars I know, tail rules consume a bounded (and often quite limited) number of tokens. This alone would give an O(N?) behaviour.

The technique presented is interesting, and can be ameliorated to the point of being usable. For example, an error state could be added to the result. It can be used as a criteria for the alternation operator (if several cases are not errors, that can be considered as an error or handled with the list of results approach given in the paper – this is better that the "use the longest match" rule used first in the paper) and to stop parsing by the sequence operator. With that modification, first, one should be able to handle non left recursive grammars (the Dragon book gives an algorithm to remove any left recursion of a grammar) and second, the exponential behaviour is removed leaving us with the quadratic one.

Removing the quadratic behaviour is easy: instead of copying a list around, use an iterator. After that, I guess that the parser will have a linear time behaviour for LL(k) grammars and be able to handle any non ambiguous grammar or any grammar with the list of results approach.

Now, how to report errors to the user in a useful way? As the method uses backtracking, it is not possible to report the error when it is detected. A possibility is to add an error message and an error position to the error state. In case of error, alternation would build an special error message if the error is that an ambiguity is detected or choose one of the error messages (the one which produced the longest parse before failing is the obvious candidate) of the failing cases when all possibilities fail.

One would still miss an error recovery scheme (that is how to continue to parse after having reported one error so that other errors can be reported), but, as I've never seen one presented for parsing techniques using backtracking, this is a very minor point.

Browsing `boost.org` I found out that they have a parser framework. From the documentation it looks like a finalized variation on the same themes as Frank's paper taking into account my comments. It surely needs to be looked at by anyone wanting to expand on Frank's paper for a purpose other than experimentation with C++ syntax or parser definition.

Jean-Marc Bourguet
jm@bourguet.org

American Hospital Supply

One of the examples given by Carr is American Hospital Supply (AHS). During the 1970s this company introduced a mainframe based electronic ordering system for hospitals. Hospitals came to value this as it allowed them to operate more efficiently and AHS increased profitability. By the 1990s other companies had similar PC based systems. Now the mainframe system became a hindrance and AHS could not compete against these competitors who used commodity systems.

For Carr this demonstrates that in the past IT could provide an advantage but, with commoditisation the advantage was eroded and became a millstone. Others could do the same thing more cheaply.

We could interpret this differently, yes AHS had an advanced system and could out-compete other firms for a while, but did it lose the advantage because of commoditisation or because it stopped development and stopped changing?

By keeping its system closed for use by itself and its customers the company gave other companies an incentive to develop alternatives. This was a management decision, not an inescapable consequence of the technology, the management could have chosen to act differently.

For example, AHS could have chosen to spin out its software development as a separate company and openly sell the software to other companies. The original AHS would still have a head start, but by selling software the company would bring in additional revenue and pre-empt the emergence of competition.

One also needs to ask: why did AHS not choose to develop a commodity version of its software? If others could then it certainly could have. This would have lowered their own cost, albeit at the expense of higher development costs. Either way, the lesson is to continue moving forward, continue to learn and move upwards with new ideas and products. So, while technology creates new opportunities it is still a servant to the management intent. At AHS management saw the intent as narrowly defined automation.

Apple Computers have faced a similar situation. Once MacOS could beat the competition hands down. but Apple didn't license it and allowed Microsoft Windows to become the commodity player. Now Apple are fighting back, precisely because their technology is not a commodity like Windows they have more control over what they do with it.

outages – and manage IT costs more closely. Why buy new PCs when the old ones still work? Why invest in storage when as much as 70% is wasted?

Is IT really a commodity?

Undoubtedly some aspects of IT have become commodities. Hard discs, RAM chips, even PCs are really commodities even if we take a perverse interest in the seek time on a Maxor versus a Shugart drive, or whether we have SIMMs or DIMMs in our box.

It is probably also true that increasingly software is becoming a commodity. Although it is an unusual commodity that is only available from one supplier, can Microsoft Word really be a commodity word processor if it is only available from one supplier? So, part of the "software as commodity" debate is entwined with the Microsoft monopoly debate.

However, it is true that mail clients, and particularly web-based e-mail are pretty much a commodity. As the ASP software model spreads software starts to look more like a service than a product.

However, there is a dimension to IT that defies the commodity classification, that is intent. This point is expressed by David Fenny:

> *"we encounter a unique characteristic of IT, its inherent lack of application purpose. If I explain to someone any of a range of traditional technologies – balance scales, bulldozers, or blast furnaces – the application is obvious. However, if I explain what is meant by a multi-media workstation, who knows what relevance it may have within a bank, a supermarket chain, or a government department." (Willcocks, 1997, p.xxii)*

We may all have access to the same commodity hardware and software but what do we choose to do with it? Two companies can buy the same hardware and software. They can each operate in the same mail order businesses but if one company intends to simply make their operation more efficient, while the other intends to identify repeat customers and sell more to them then there is a significant different in the outcome. Of course, this increases the importance of getting your implementation and roll-out right.

The true power of IT is beyond simple automation and efficiency, that is a commodity. If we want to get the most from IT we need to use it in innovative ways and keep innovating. IT becomes a tool to help bring about change, and indeed, learn to do things better than our competition. And most importantly of all, keep innovating, changing and learning. Not that we want to change for change's sake – or for the sake of the IT – but, change for the sake of the businesses.

IT has another agenda

The role of IT has traditionally been seen as one of automation. Sometimes, when you speed things up enough you get something new. For example, you could view Amazon as a very fast form of catalogue shopping, but it is so fast that is has become a new way of shopping.

However, IT has another agenda, one of learning. Imagine being asked by one of your company's clerks, Bert, for a small application. He explains what he wants to you – so you learn something. But at the same time you ask questions "Why do you do it like that?" which forces Bert to think and learn himself.

Next you then code the application and show it to Bert. As a result you are both forced to think, together, on what it is doing, to remove incorrect assumptions and even improve the entire process. You have both learned.

Now the application is deployed Bert has some spare time on his hands. So he can follow up some of those complaints (yes, the ones he was throwing in the trash). As a result he talks to Doris and together they realise that if we could just extend the application a bit, it could help Doris and cut down on some of the mistakes.

(Unfortunately, with all this done your boss notices that he doesn't need both Bert and Doris so fires one of them. The next week he is told to cut his IT budget so he fires you too.)

The point is: IT has the power to help people learn about their business not because it provides some neat little training package, but because it helps us reflect on what we are doing and why. If handled correctly the process of introducing IT helps us remove obstacles which block our vision and encourages us to think about the bigger picture.

Not only is it bringing about learning but it brings about change. Sometimes for the better, and sometimes for the worse, but if we simply automate what already exists, then we don't see the full benefit of IT.

This is where we leave the realm of IT and consider management. If management don't want change then fine, things can stay as they are, but ultimately someone else will adopt the changes we reject and beat us in business.

If management accept change then there are two ways to go about it. One is the top-down, mandated change that we saw with the business process re-engineering (BPR) movement. This is the change that says "The consultant knows best, there shall be an IT system and this is what you shall do." This has the capacity to destroy businesses.

Alternatively, there is the more compassionate management who want to harness this change and learning for the benefit of the business. Giving Bert and Doris their new system improves the quality of their work, recognising that the people who work with the existing system probably know more about the subtleties in the process than a BPR consultant ever will.

So far, I've described this from a business perspective. The flip side, the IT perspective, is also interesting. If you try to introduce a new system without properly considering those who will use it then you will encounter problems. And if you've ever wondered why people tell you something today, and come to you tomorrow to contradict themselves it may be that in telling you they too were learning.

In both perspectives we are uncovering knowledge. Such knowledge can lie unrecognised until IT is applied to the problem, but, while the IT may be a commodity and offer no competitive advantage this is not true of the knowledge. Indeed, knowledge offers a very special resource that can be used to give companies a unique competitive advantage.

Move on up

There is another reason why software need not become a commodity. As we complete software and hardware technologies we raise our sights, we tackle bigger problems.

There was a time when developing a new computer meant developing a new operating system. We still develop new OSs, but if we are building a new machine we won't need an army of developers to write an OS. We can buy Windows off the shelf, or port Linux. This frees the resources (money and developers) to concentrate on new applications.

A good example of this is XML. Ten years ago all file formats were proprietary, getting data from Lotus 1-2-3 into my applications was painful. The idea of getting it in nicely tagged mark-up language would have been a joy. Problem solved.

But now that we have XML, and the file format problem is solved we don't stop. EDI (electronic data interchange) is being re-invented, web services are appearing, SOAP is being used in applications where we would never have dreamed of using CORBA or DCOM.

XML may have solved one immediate problem, but in doing so we created a thousand other opportunities for using it. Indeed, we are still learning of new applications for XML.

In short, as we commoditise one part of the software market it serves as a base to move onto the next. Again, we are learning, always trying something just beyond our reach. It doesn't matter if today's advantages are tomorrow's commodities, we will have moved on to some new advantage.

What does this mean for software development teams?

On the one hand, if Carr is right it doesn't look like there is much future for software development teams. However, if we view software as the medium in which we embed our knowledge then there is a brighter future.

In this future software developers codify company knowledge. They become what Nonaka calls Knowledge Engineers. Of course, the software is not itself knowledge but it is the result of knowledge work. By changing, or not changing, the software developers become the gatekeepers of change. Choosing to accept a change request will spread one person's insights to many, refusing the same request is limiting our capacity to change.

There is a very difficult line to define here between what changes should be accepted and what should not. This is nothing new in software development but it does mean we need to move away from the myth that we can limit change. Forget the idea that if we had enough time, enough analysis and good enough people we could write down a complete specification. Forget it because the very process of writing it down will change it.

Your best analyst could work with Bert for six months and produce the perfect specification. However, as soon as people, and especially Bert, see it coded they will learn and see room for improvement.

Fortunately, software people are used to change. We love learning new languages, operating systems, and application domains. Unfortunately, we don't always recognise that other people don't relish change in quite the same way, in truth most people don't like change and feel threatened by it.

I increasingly suspect the reason IT people have a reputation for lacking social skills is simply that they are placed in the position of introducing change where people don't want it. My suspicion is that the social skills of IT people are at least average, but introducing change requires more understanding and empathy than average. Not only this, but we often work under time constraints that don't leave us time to talk through someone's problems, or sympathise with them.

Conclusion

If Carr is right, and we want to stay in the software business we have two choices. We either need to get into writing commodity packages, or we need to accept a life in maintenance.

However, I can't accept Carr's argument. I think he is exposing an over-simplistic view of IT, for all the reasons I've outlined above I think he's wrong. And for all the same reasons I increasingly believe we need to view software less as IT and more as business.

Allan Kelly

References

[1] Carr, N.G., 2003, "IT Doesn't Matter", *Harvard Business Review*, May 2003. Available at `www.hbr.com`, also check his website, `http://www.nicholasgcarr.com`, where you will find some other responses to his ideas.

[2] Nonaka, I., and Takeuchi, H., 1995, *The Knowledge Creating Company*, Oxford University Press

[3] Willcocks, L., Feeny, D., and Islei, G., 1997, *Managing IT as a Strategic Resource*, McGraw-Hill

Statically Checking Exception Specifications

by Ken Hagan

The C++ newsgroups occasionally have threads about “fixing” exception specifications (hereafter “ES”) so that they can be checked at build-time. One practical problem is maintenance; an ES depends on all the callees of a function as well as the function itself. A more fundamental problem is that the exceptions that can be thrown from templated code can vary with the instantiation parameters and these are clearly unknowable when the programmer writes the template. Perhaps the programmer is the wrong person to be writing the ES.

Outline of a Solution

As the language stands right now, a function with no ES can throw anything. My basic change is to say that when a function has no explicit ES, the programmer wants the build system to deduce one.

Except where dynamic linking is used, the compiler can determine the most restrictive ES that the programmer could have written for the function, by noting which exceptions are thrown and caught and which functions are called. Since source code isn’t generally available for the called functions, the calculation cannot be completed, but it can be reduced to an ES-expression. For example, for this function the compiler might emit `ES(Foo)=ES(Baz)-Quux+Oink`.

```
void Foo() {
  try { if(Baz()==42) throw Oink(); }
  catch (Quux& b) { /*stuff*/ }
}
```

The linker then considers each function in turn, replacing expressions with an absolute ES wherever possible. If not every expression is resolved on the first pass, it makes another pass and so on until completed. When compiling templates, the compiler can emit ES-expressions that depend on template parameters. When instantiating those templates (perhaps in a pre-link phase) the expressions can be converted to non-dependent expressions.

The ES-expressions may be stored in a separate file, in the object file using some extension, or in the object file as an un-nameable data item that the linker is sure to discard. The first is cumbersome, the second might conflict with an ABI and the third is a filthy hack, but all three are workable. For static libraries, library code is no different from our own as far as the linker is concerned.

To eliminate false positives we need a new cast: the `nothrow_cast`. It operates on pointers to functions; so it modifies individual calls rather than the definitions. It tells the compiler that this invocation of the function will not throw the specified type. As usual with casts, if you lie to the compiler then it will get its revenge in the form of implementation defined or undefined behaviour.

Four complications

1) Pointers to Functions

We expect to be able to write an expression for the minimal ES involving only class names and the ES-es of named functions. With function pointers, we don’t know which function they point to. We can, however, identify the pointer itself. It must be one of the following 4 cases, and each can be named.

- A global variable, such as `baz` (in the example below).
- A struct or class member, such as `Quux::m_pfn`.
- A function parameter or return value, such as `Foo(4th)`.
- A local (automatic) variable, such as `Foo()::name`.

For structure members, and function parameters, no attempt is normally made to distinguish between different instances or invocations. The local variable case can be eliminated by the compiler because `ES(Foo()::name)` can always be replaced with the ES of whatever was used to initialise `name`, but that’s just an optimisation.

What then? Well firstly, we can write the minimal ES for a function that uses such a pointer, simply referring to the ES of this named item.

```
extern int (*baz)();
void Foo() {
  try { if((*baz)()==42) throw Oink(); }
  catch (Quux& b) { }
}
```

For this function, `ES(Foo) = ES(*baz)-Quux+Oink`. Great, as long as the linker can figure out `ES(*baz)`. That’s slightly harder than `ES(Baz)`, because, informally, `Baz` is a constant but `baz` is a variable. However, we can model `(*baz)()` as a function that calls all of the functions that are assigned to `baz` throughout the program, and each of those assignments will be seen by the compiler. For each assignment, the compiler can spit out `ES(*baz) += ...` , where the right hand side is either `ES(Function)` or `ES(*another_pointer)`.

All the linker has to do is join all the pieces together. The linker has an ES for a named object that possibly depends on the ES-es of other named objects. Pointers to functions are now no different from the functions themselves, and they all get thrown into the pot and resolved together.

Pointers to pointers to functions, such as virtual function tables, add little new to the problem. Instead of tracking everything that `(*p)()` might point to we have to track everything that `(**pp)()` might point to.

2) Recursion

In the presence of recursion, the call graph of a program has cycles. For such functions we might have `ES(Foo) = +Quux - Oink + ES(Foo)`. Now, a function neither increases nor decreases its ES by calling itself, so when `ES(Foo)` appears on the right hand side of an expression *for itself* it can be ignored. This allows us to break the cycles in recursive systems.

Though we can ignore `ES(Foo)` for itself, we cannot ignore it elsewhere in the cycle. Part of the cycle might throw exceptions that are caught by other parts of the cycle, so the minimal ES exposed from a recursive cycle depends on where you enter it.

In fact, this property that lets us evaluate ES-expressions in any order we like. We can just pick one and recursively replace every term on the right hand side with its expansion. Eventually we will have a long expression with either absolute classes or repetitions of the left-hand side. We remove the latter and we have our absolute ES.

3) Shared Libraries

For shared libraries, the linker doesn’t see the actual code when it is linking the client application. Whether this is a problem depends on how the linking is achieved. I’m familiar with Windows, so I’ll treat the two cases on that platform and then ignorantly assert that other platforms add nothing new.

The first case is load-time dynamic linking. The linker is given an "import library" which describes the data and functions exposed by the DLL. Any references to those are replaced with placeholder items in the linked application and the operating system loader "fills in the blanks" when the application is loaded into memory to run. The provision of an import library makes this case very similar to the static case. I believe it is sufficient for the import library to contains ES-expressions for just those items mentioned in the library's header file, since that it all the compiler sees and so that is all that can appear in client object files. This is certainly the case in my extended example.

The second case is run-time linking, where the program uses some magic to conjure an address out of the ether. To take a slightly non-trivial example...

```
extern IFoo* CreateSuperFoo();
                     // in external library

IFoo* (*pfn)() = /*magic*/;
                     // in client code
IFoo* p = (pfn)();
```

The details of `CreateSuperFoo` and also of whatever `IFoo`-derived class that this library actually offers is a complete mystery to the build system. It may be written in a different language so it is quite possible that neither the compiler nor linker ever see it. Here we have the one place where I think a programmer has to write an ES.

The two main objections to ES that I noted at the beginning of the article don't apply. A dynamically loaded extension cannot be a template, though it may be an instantiation. Neither is it likely to change often and even if it does, all knock-on effects on the rest of the system are now the compiler's problem.

4) False Positives

The final problem is that a function might throw an exception in the case of bad input, and carry an ES to that effect, but many callers might never feed bad parameters into the function. This partly depends on one's programming style. If I might return to the example...

```
if(x<0) return false;
x = Sqrt(x); // assuming Sqrt() throws
             // when x<0
```

There are certainly situations where one should write one of the following...

```
Type Sqrt(Type x) { if(x<0) abort(); ... }
Type Sqrt(Type x) { assert(x>=0); ... }
```

...and happily spit in the faces of irate clients whose programs were aborted, saying "Don't do that then!". However, if we choose to throw an exception instead then our clients will either be faced with link-time ES errors or be forced to write such abominations as

```
if(x<0) return false;
try { x = Sqrt(x); } catch (...) {
/*unreachable*/ }
```

Not only does this look bad, but it probably incurs run-time penalties (space and time). As with the function pointers, we know something that the compiler doesn't, so we tell it with a cast.

```
if(x<0) return false;
nothrow_cast<std::logic_error>(Sqrt)(x);
```

The `nothrow_cast` tells the compiler that the function does not throw the mentioned type.

Costs and Benefits

I think it is worth confessing at this point that I've only spent the time and energy on this because I wanted static checking. Showing that it could be done with minimal impact on existing source code seemed like a good way to argue the case. It all turned out a little harder than I expected, so is static checking worth this effort?

First, I note that the current standard allows ES violations at run-time, so any ES violations detected by this system can only result in linker warnings. The linker must still generate a working executable.

Costs and Limitations

The scheme derives the minimal ES from whatever source code is presented to it, so the same program might "fail" if compiled against StlPort rather than Dinkumware, or if compiler settings change. A debug build might give false positives that an optimising build can rule out as a by-product of its analysis.

You and your library vendors will all have to run all the code through the new compiler. The scheme adds no new compile-time errors, so if the library vendors are still in business then they shouldn't have much of a problem with this.

If you don't modify the code then you may get warnings from the ES checking phase, which will disable the various optimisations mentioned below. You've lost nothing except for the extra build costs.

If you are able to modify your code, you can eliminate all the errors using explicit ES and `throw_cast`, respectively. In both cases you can let the diagnostics guide you. There is no problem of figuring out what changes to make, simply the time involved in actually doing it.

The extension does require more complicated compilers and linkers. I can't judge how much more complicated because I've never written a compiler or linker, let alone one for C++. There is also a cost in build time which I don't feel qualified to estimate, but I have already noted that we don't need to reduce ES expressions in any particular order.

Benefits

Having to treat ES violations as warnings actually yields a couple of migration paths. A vendor could just ignore the whole idea, implementing the `nothrow_cast` as a do-nothing template function. Equally, since there is no new run-time behaviour, the whole thing could be done by a tool like lint.

If we can spot violations of `throw()` at build-time rather than run-time, with any tool, the Abrahams exception safety guarantees are easier to police. The cast may be useful to the compiler even without the link-time checking, since it can optimise more strongly if it believes exceptions can't happen.

However, we get maximum benefit if the compiler and linker do the checking. The cost of exceptions that cannot ever occur can be reduced to zero in both time and space and any function that can't throw (and all its immediate callers) can be recompiled with that knowledge. With these optimisations, Standard C++ is more attractive for embedded systems and vendors needn't include compiler options to disable exceptions.

Ken Hagan

Consider the common scenario of an interface header file...

```
  struct IFoo {                // struct IFoo::vbtl {
    virtual void Bar() = 0;    //    void (*pBar)(IFoo*);
    virtual int Quux() = 0;    //    int (*pQuux)(IFoo*);
  };                           // };
  void AddFoo(IFoo*);
  void DoStuff();
```

...used by a shared library source file...

```
  IFoo* global;                // IFoo::vtbl** global;
  void AddFoo(IFoo* foo)
    { global = foo; }          // ES(AddFoo) = 0
                               // ES(*IFoo::vtbl.pBar) += ES((*AddFoo 1st).pBar)
                               // ES(*IFoo::vtbl.pQuux) += ES((*AddFoo 1st).pQuux)
  void DoStuff()
    { global->Bar(); }         // ES(DoStuff) = ES(*IFoo::vtbl.pBar)
```

...and implemented in an application source file...

```
  class Foo : public IFoo {
    virtual void Bar()         // ES(*IFoo::vtbl.pBar) += ES(Foo::Bar)
      { throw 1; }             // ES(Foo::Bar) = int
    virtual int Quux()         // ES(*IFoo::vtbl.pBar) += ES(Foo::Bar)
      { return 0; }            // ES(Foo::Quux) = 0
  };
  int main() {
    AddFoo(new Foo);           // ES( *(*(AddFoo 1st).pBar) ) += ES(Foo::Bar)
                               // ES( *(*(AddFoo 1st).pQuux) ) += ES(Foo::Quux)
    DoStuff();
  }                            // ES(main) = ES(AddFoo) + ES(DoStuff)
```

In the application, the compiler can see that the `IFoo*` parameter to `AddFoo` is actually a "`new Foo`". Had that detail not been visible, the compiler could only have written...

```
  ES( *(*(AddFoo 1st).pBar) ) += ES(*IFoo::vtbl.pBar)
  ES( *(*(AddFoo 1st).pQuux) ) += ES(*IFoo::vtbl.pQuux)
```

We bring all this together in the linker. Our raw data from compiling the library is...

```
  ES(*(IFoo::vtbl->pBar)) += ES(*(AddFoo 1st)->pBar)
  ES(*(IFoo::vtbl->pQuux)) += ES(*(AddFoo 1st)->pQuux)
  ES(DoStuff) = ES(* IFoo::__vtable.pBar)
  ES(AddFoo) = 0
```

That from compiling the application is...

```
  ES(*IFoo::vtbl.pBar) += ES(Foo::Bar)
  ES(Foo::Bar) = int
  ES(*IFoo::vtbl.pQuux) += ES(Foo::Quux)
  ES(Foo::Quux) = 0
  ES(*(AddFoo 1st)->pBar)) += ES(*IFoo::vtbl.pBar)
  ES(*(AddFoo 1st)->pQuux)) += ES(*IFoo::vtbl.pQuux)
  ES(main) = ES(AddFoo) + ES(DoStuff)
```

Bringing it all together and substituting yields...

```
  ES(main) = 0 + ES(DoStuff)
           = ES(*(AddFoo 1st)->pBar)
           = ES(*IFoo::vtbl.pBar)
           = ES(Foo::Bar)
           = int
```

Software As Read

by Jon Jagger

Programming is writing, and writing is visual. We should explore software as read not code as executed. Less code, more software.

Iteration

In his Overload 45 (October 2001) article, *minimalism – omit needless code*, Kevlin worked on the simple problem of printing the `std::string`s inside a `std::vector` to `std::cout`. An early version looked like this:

```
typedef vector<string> strings;
typedef strings::iterator iterator;
for (iterator at = items.begin();
     at != items.end(); ++at) {
  cout << *at << endl;
}
```

A later version looked like this:

```
class print { ... };
for_each(items.begin(), items.end(),
         print(cout));
```

And the final version looked like this:

```
typedef ostream_iterator<string> out;
copy(items.begin(), items.end(),
     out(cout, "\n"));
```

Readability

The main source of repetition is repetition. When programming in C++ you often find yourself making a call to a template function where two of the arguments are created by calling `begin` and `end` on a container. This quickly gets repetitive. The repetition itself suggests several solutions. Ranges are basic building blocks of the STL design and it is surprising they are not a visible and explicit artefact of its type system. For example:

```
template<typename iterator>
class range {
public: // types
  typedef iterator iterator;
public: // c'tor
  range(iterator start, iterator finish)
    : from(start), until(finish) { }
public: // properties
  iterator begin() const {
    return from;
  }
  iterator end() const {
    return until;
  }
private: // state
  iterator from, until;
};
```

This would make containers substitutable for a range over themselves which would in turn allow STL algorithms to expect a range argument rather than two iterator arguments. For example:

```
template<typename range,
         typename function>
function for_each(const range & all,
                  function apply) {
  return for_each(all.begin(), all.end(),
                  apply);
}
```

This version of `for_each` is not part of STL so you have to provide it yourself. Once you've done this you can rewrite this:

```
for_each(items.begin(), items.end(),
         print(cout));
```

as the impressively readable:

```
for_each(items, print(cout));
```

Understanding this statement is a complete no brainer. It clearly and concisely expresses its intention. However, it does require you to create the `print` class (which hides away the "`\n`" detail). Alternatively, you could pull the same trick by writing a non standard version of `copy`:

```
template<typename range, typename output>
output copy(const range & source,
            output sink) {
  return copy(source.begin(),
              source.end(), sink);
}
```

allowing the beautifully readable:

```
copy(items, out(cout, "\n"));
```

Preference

Which versions do you prefer? The explicit iteration, the `for_each` versions, or the `copy` versions? Can you explain why?

I prefer the `copy` versions. The name `for_each` is itself a subtle but strong hint that iteration is involved. It suggests that each of the items will be printed to `std::cout`, one at a time. The iteration comes first (`for_each`, leftmost), followed by the action (`print`, rightmost). In contrast, the `copy` is subtler and simply suggests copying the items to `cout`. It has more of a "single operation" feel to it. The iteration is not visible (and the "`\n`" is). This difference is important, not because you should always try to hide all iteration, but because the intention was to "write the items to `cout`". In other words, the `copy` version is a simpler and more direct expression of the problem. Lots of code is too solution focused; it lacks an expression of the problem and hence is hard to understand and costly to maintain.

Many thanks to Kevlin for an insightful review of a first draft of this article.

Jon Jagger
jon@jaggersoft.com
www.jaggersoft.com

Chaos Theory – Part 2

by Alan Griffiths

In a couple of ways this article represents a return to my past: both to C++ and to the “Chaos Theory” theme. For the last couple of years my professional interest has been diverted from C++ to other areas (specifically to Java, J2EE and development methods). As a result I’ve accumulated a backlog of C++ related material waiting to be read. In particular, I’ve finally found time to read “Modern C++ Design” which demonstrates the ability to use the language to do things at compile time. Other books (like “Generative Programming”) that I’ve read during my diversion have also used these ideas and there are libraries (boost has a fine example) to support these uses. But I wanted to do more than read and admire these novel ideas. I wanted to try them out – but I was in search of a problem.

The problem I chose is one that I wrote about once before – in the “first” article in a series of articles on “Chaos Theory”. This was a long time ago, I can’t remember why but the rest of the series never materialised (in fact I can’t find a copy of the first article either – but I think it was published in C Vu about ten years ago.)

Chaos theory is a branch of mathematics that was developed in the nineteenth century by Poincaré in an attempt to solve the problem “Is the Solar System stable?” Although he failed to solve the problem he made a sufficient dent to be awarded a significant prize for this work. Towards the end of the last millennium work on the stability of mathematical systems grew in importance with the increasing use of computers to do numerical modelling.

The types of mathematical model to which chaos theory applies are those that develop over time and whose current state depends upon the past. It gets interesting when this change is complicated enough that exact solution is infeasible and numerical modelling is the only approach to getting results. What the mathematicians showed was that even when it isn’t possible to write down an exact description of the evolution of the model it is still possible to make useful predictions about the type of behaviour.

This sounded fun, so I decided to try it for myself and started writing a series of articles for C Vu. At least I think I did – I never wrote the second article and I can’t find the first! The first article introduced an easy to understand non-linear system and demonstrated the application of these predictions. The system in question takes a pair of numbers and generates a new pair of numbers – and what chaos theory predicts is that one of three things will happen:

1 There is an infinite non-repeating sequence of number pairs.
2 Eventually the sequence of number pairs settles into a limited range of values – an “attractor”.
3 From some point in the sequence all the number pairs have the same value. (This is really a special case of 2)

(In the particular system I’m writing about the first of these is extremely implausible and it turns out that case 3 is what happens.) My thoughts turned to finding these fixed values at compile time.

```
/*
  "This sentence has eight vowels and
   twenty consonants"

The above sentence is false because the
numbers eight and twenty are arbitary
(and wrong).  But we can create a
sequence of number pairs (Vn, Cn) by
substituting the numbers into a sentence
of this form and counting the vowels and
consonants to get the next pair of
numbers.
  Continuing to substitute these values
back into the sentence then one of three
things must happen:

1/ The series of pairs (Vn, Cn)
   diverges
2/ The series of pairs (Vn, Cn)
   loops though a sequence of values
3/ The series of pairs (Vn, Cn)
   converges to constant values

The following program executes this
algorithm *at compile time* to find
values of (Vn, Cn) which make the
sentence true and outputs the
result.
*/

#include <string>
#include <iostream>
namespace {

/*
The first issue to address is that it
isn’t possible to count vowels or
consonants in a string at compile
time. Compile time processing is
limited to creating types and
constant integral expressions. There
are two approaches to this that
occur to me: create a type for each
character and represent a sentence
as a typelist or break the sentence
into subsentences representing the
fixed and variable portions and
represent these as types. While
the former is clearly a lot more
general it involves more work and
it is the latter approach to the
problem that I adopted.
  So for the variable parts I have:
*/

template<int count>
struct number_as_subsentence;

/*
OK, we’ll have to define some
specialisations for this before we can
use it, but this is a useful placeholder
for the full sentence template.
*/
```

```
template<int vowels, int consonants>
struct sentence {
  enum {
    no_of_vowels = 11
      + number_as_subsentence<vowels>
                          ::no_of_vowels
      + number_as_subsentence<consonants>
                          ::no_of_vowels,

    no_of_consonants = 23
      + number_as_subsentence<vowels>
                      ::no_of_consonants
      + number_as_subsentence<consonants>
                      ::no_of_consonants,
    is_true = no_of_vowels == vowels
        && no_of_consonants == consonants
  };

  static std::string as_string() {
    static const std::string
         beginning("This sentence has ");
    static const std::string
         middle(" vowels and ");
    static const std::string
         end(" consonants!");

    return beginning
      + number_as_subsentence<vowels>
                            ::as_string()
      + middle
      + number_as_subsentence<consonants>
                            ::as_string()
      + end;
  }
};

/*
This template provides a compile time
mechanism to take a number of vowels
and a number of consonants and determine
the effect of placing them into our
template for a sentence. It will also
construct the corresponding sentence
for us.
  What’s next? In the original
program there was a loop to keep
trying the sequence of sentences
until we find one that is true. But
that is another thing that we cannot
do at compile time: we can’t do
iteration, we have to rework the
algorithm as recursion:
*/

template<int vowels,
         int consonants,
         bool finished = false>
struct calculate_sentence
       : private sentence<vowels,
                          consonants> {
  typedef typename calculate_sentence<
        calculate_sentence
                      ::no_of_vowels,
        calculate_sentence
                      ::no_of_consonants,
        calculate_sentence::is_true>
                      ::result result;
};

/*
This keeps trying new sentences all
right, but we need to end the recursion
(which is what the third parameter is
for).  Interestingly one cannot use a
“metaprogramming if_” (like that in the
boost library) here because both the true
and false conditions get instantiated.
With the current approach we just need a
specialisation of calculate_sentence as
follows:
*/

  template<int vowels, int consonants>
  struct calculate_sentence<vowels,
                     consonants, true> {
    typedef sentence<vowels,
                     consonants> result;
  };

/*
That is really all the interesting bits
of the program done. The templates for
describing the numbers are tediously
repetitive - but there is a useful pre-
processor to handle that:
*/

#define NUMBER_AS_SUBSENTENCE(number,\
              text, vowels, consonants)\
  template<>\
  struct number_as_subsentence<number> {\
    static std::string as_string() {\
      return  text;\
    }\
\
    enum {\
      no_of_vowels = vowels,\
      no_of_consonants = consonants\
    };\
  }

  NUMBER_AS_SUBSENTENCE(0,"zero",2,2);
  NUMBER_AS_SUBSENTENCE(1,"one",2,1);
  NUMBER_AS_SUBSENTENCE(2,"two",1,2);
  NUMBER_AS_SUBSENTENCE(3,"three",2,3);
  ...
  NUMBER_AS_SUBSENTENCE(48,"forty eight",3,7);
  NUMBER_AS_SUBSENTENCE(49,"forty nine",3,6);
  NUMBER_AS_SUBSENTENCE(50,"fifty",1,4);
```

[concluded at foot of next page]

Single Exit

by Jon Jagger

In CVu 15.4 Francis makes a case that some functions are less complex if they use multiple return statements. In Overload 55 I stated my preference for single exit via a single return. I'd like to explore the examples Francis presented to try and explain my preference more explicitly.

Example 1 – Multiple Returns

The first code fragment Francis presented was as follows:

```
bool contains(vector<vector<int> >
             const & array, int value) {
  int const rows(array.size());
  int const columns(array[0].size());
  for (int row(0); row != rows; ++row) {
    for (int col(0);
         col != columns;
         ++col) {
      if (array[row][col] == value) {
        return true;
      }
    }
  }
  return false;
}
```

And he wrote:

"if you are a believer that functions should never have more than a single return you have a problem because however you reorganise your code the requirement is for two distinct exit conditions".

I'm a firm believer, but even if I wasn't I'd have to agree that some multiple returns somehow feel more acceptable than others. And as Francis says *"perceived complexity is a function of many things"*. However I don't think it's quite accurate to say the requirement is for two distinct exit conditions. To try and explain what I mean, consider the following implementation of `contains`[1]:

```
bool contains(const vector<vector<int> >
                    & array, int value) {
  vector<int> all;
  for (int at = 0; at != array.size();
       ++at) {
    copy(array[at].begin(),
         array[at].end(),
         back_inserter(all));
  }

  return find(all.begin(), all.end(),
              value) != all.end();
}
```

This is an unusual implementation but it does show that the requirement is always to return a single value to the function caller (in this case either `true` or `false`). Exactly how you do so depends on your choice of implementation which is a different matter. Another approach would be to design an iterator adapter class that "flattens" the iteration through a container of containers.

Francis continues *"The only ways these can be combined in a single return statement require either continuing processing after you know the answer or increasing the perceived complexity of the code."* Here is the heart of the issue – the complexity of the code. Is a single-return version of `contains` necessarily more complex?

Example 1 – Single Return

Here's a more realistic single-return version of `contains`:

```
bool contains(const vector<vector<int> >
                   & values, int value) {
  int at = 0;
  while (at != values.size()
         && !exists(values[at], value)) {
    ++at;
  }
  return at != values.size();
}
```

This makes use of the following non-standard helper function:

```
template<typename range, typename value >
bool exists(const range & all,
            const value & to_find) {
  return find(all.begin(), all.end(),
              to_find) != all.end();
}
```

1 The array[at] duplication can be avoided like this:

```
copy(array[at], back_inserter(all));
```

which uses a handy (but sadly non-standard) version of copy which, coincidentally, I also used in my other article (Software As Read).

```
template<typename range, typename output>
output copy(const range & source,
            output sink) {
  return copy(source.begin(), source.end(),
              sink);
}
```

[continued from previous page]

```
  #undef NUMBER_AS_SUBSENTENCE
}

/*
To run the program we only need
instantiate the template and output the
result...
*/
```

```
int main() {
  std::cout << calculate_sentence<8, 20>
               ::result::as_string()
            << std::endl;
}
```

Alan Griffiths
alan@octopull.demon.co.uk

The full program is available on my website at:
http://www.octopull.demon.co.uk/C++/ThisSentence/

Example 1 – Comparison

What are the differences between these single/multiple return versions?

- **Line count.** No difference. (I haven't counted lines containing a single left/right brace).
- **Maximum indent.** The deepest control in the multiple-return version is 3 – the return in an `if` in a `for` in a `for` whereas the deepest control in the single-return version is 1 – the increment in the `while`. This is the reason the single-return version needs fewer lines containing a single left/right brace.
- **Function count.** The multiple-return version is a single function whereas the single-return version uses a helper function. The helper function is useful in its own right and could quite conceivably have already existed. Small helper functions are significant because they can help to make other functions smaller and clearer.
- **Loop scaffolding complexity.** By using the helper function the single-return version has lost a whole level of iteration scaffolding.
- **Return expression complexity.** The multiple-return version uses two literals, `true` and `false`. One of these returns occurs at indentation level 3. In contrast the single-return version uses a single boolean expression at indentation level 1.
- **Loop condition complexity.** The multiple-return version has two very simple (and very similar) boolean expressions as its two `for` statement continuation conditions. The single-return version has one (more complex) boolean expression in its `while` statement continuation condition. How comfortable you are with this more complex boolean expression (using the && short-circuiting operator) is largely a matter of how familiar you are with this style.
- **Style.** If you are looking for an element in a C++ vector you could argue that it's reasonable to expect to use the C++ `find` algorithm, as the multiple-return version does. In contrast the single-return version uses a more C-like explicit subscript iteration. The difference is quite subtle in this case but it does serve to highlight an important point Francis made – *"perceived complexity is a function of many things (one of them being the individual reader)"*. I think its fair to say the more experience you have of "mature" C++/STL style the more readable you'd find the single-return version.

Example 2 – Multiple Returns

The second code fragment Francis presented is as follows (some code elided, I assume the `int <-> bool` conversions are deliberate):

```
bool will_be_alive(life_universe
                           const & data,
                   int i,
                   int j) {
  int const diagonal_neighbours = ...;
  int const orthogonal_neighbours = ...;
  int const live_neighbours =
                  diagonal_neighbours +
                  orthogonal_neighbours;
    if (live_neighbours == 3)
      return true;
    if (live_neighbours == 2)
      return data[i][j];
    return false;
  }
```

I would start by rewriting this as follows:

```
bool will_be_alive(const life_universe &
                                    data,
                   int i,
                   int j) {
  const int diagonal_neighbours = ...;
  const int orthogonal_neighbours = ...;

  const int live_neighbours =
                diagonal_neighbours +
                orthogonal_neighbours;

  if (live_neighbours == 3)
    return true;
  else if (live_neighbours == 2)
    return data[i][j];
  else
    return false;
}
```

The difference is the explicit coding of the control-flow surrounding the return statements. Do you think making the control-flow explicit is a good thing? If you're not that bothered I invite you to consider the following:

```
if (live_neighbours == 3)
return true;
```

I hope you're more concerned by this lack of indentation. These days indenting your code to reflect logical grouping is taken as an article of faith that people forget to question or recall exactly why it is used. Indentation visibly groups similar actions and decisions occurring at the same level. If you believe that indentation is a Good Thing there is a strong case for *clearly* and explicitly emphasising that all three return statements exist at the *same* level. In contrast, and significantly, the multiple-returns in the first example are not at the same level.

Example 2 – Single Return

Francis also presented example 2 using a single return involving nested ternary operators:

```
return (live_neighbours == 3)
  ? true
  : (live_neighbours == 2)
    ? data[i][j]
    : false;
```

I agree with Francis that this adds nothing in terms of clarity. In fact I think it's a big minus. This is the kind of code that gives the ternary operator a bad name. But inside this long and inelegant

[concluded at foot of next page]

Error and Exception Handling

by David Abrahams

When should I use exceptions?

The simple answer is: "whenever the semantic and performance characteristics of exceptions are appropriate."

An oft-cited guideline is to ask yourself the question "is this an exceptional (or unexpected) situation?" This guideline has an attractive ring to it, but is usually a mistake. The problem is that one person's "exceptional" is another's "expected": when you really look at the terms carefully, the distinction evaporates and you're left with no guideline. After all, if you check for an error condition, then in some sense you expect it to happen, or the check is wasted code.

A more appropriate question to ask is: "do we want stack unwinding here?" Because actually handling an exception is likely to be significantly slower than executing mainline code, you should also ask: "Can I afford stack unwinding here?" For example, a desktop application performing a long computation might periodically check to see whether the user had pressed a cancel button. Throwing an exception could allow the operation to be cancelled gracefully. On the other hand, it would probably be inappropriate to throw and handle exceptions in the inner loop of this computation because that could have a significant performance impact. The guideline mentioned above has a grain of truth in it: in time critical code, throwing an exception should be the exception, not the rule.

How should I design my exception classes?

1 Inherit from `std::exception`. Except in *very* rare circumstances where you can't afford the cost of a virtual table, `std::exception` makes a reasonable exception base class, and when used universally, allows programmers to catch "everything" without resorting to `catch(...)`. For more about `catch(...)`, see below.

2 Don't embed a `std::string` object or any other data member or base class whose copy constructor could throw an exception. That could lead directly to `std::terminate()` at the throw point. Similarly, it's a bad idea to use a base or member whose ordinary constructor(s) might throw, because, though not necessarily fatal to your program, you may report a different exception than intended from a throw-expression that includes construction such as:

```
throw some_exception();
```

There are various ways to avoid copying string objects when exceptions are copied, including embedding a fixed-length

---

[continued from previous page]

statement there is a shorter and more elegant one trying to get out. To help it escape consider a small progression. We start with this (not uncommon) pattern:

```
bool result;
if (expression)
  result = true;
else
  result = false;
return result;
```

This is exactly the kind of code that gives single-exit a bad name. It is overly verbose; it isn't a simple, clear, and direct expression of its hidden logic. It is better as:

```
if (expression)
  return true;
else
  return false;
```

But this is still overly verbose. So we take a short step to:

```
return (expression) ? true : false;
```

And removing the last bit of duplication we finally arrive at:

```
return expression;
```

This is not better merely because it is shorter. It is better because it is a more direct expression of the problem. It has been stripped of its solution focused temporary variable, its `if-else`, and its assignments; all that remains is the problem focused expression of the answer. It has less code and more software. Applying the same process to the chained `if-else` containing three return statements we arrive not at a nested ternary operator but at this:

```
return live_neighbours == 3 ||
       live_neighbours == 2 &&
       data[i][j];
```

This is focused on and is a direct expression of the problem in exactly the same way.

Conclusion

My rules of thumb are as follows:

- Almost all functions are better with a single return. The issue is separation of concerns. Do you separate out the statements that determine the answer from the statement/s that return the answer? Multiple-return versions don't whereas single-return versions do.
- Multiple return statements become less acceptable the further apart they become (both in terms of logical indentation and physical line number). Large functions have greater scope for abuse simply because they allow multiple returns to live farther apart.
- Multiple return statements are more acceptable when they are all at the same level of a mutually-exclusive selection. In most cases these multiple returns can be refactored into a more expressive single return.

But remember, dogmatically following rules is not a recipe for good software. The best software flows from programmers who think about what they do and who follow principles and practices that naturally generate quality.

Many thanks to Kevlin for an insightful review of a first draft of this article.

Jon Jagger
jon@jaggersoft.com

buffer in the exception object, or managing strings via reference-counting. However, consider the next point before pursuing either of these approaches.

3 Format the `what()` message on demand, if you feel you really must format the message. Formatting an exception error message is typically a memory-intensive operation that could potentially throw an exception. This is an operation best delayed until after stack unwinding has occurred, and presumably, released some resources. It's a good idea in this case to protect your `what()` function with a `catch(...)` block so that you have a fallback in case the formatting code throws

4 Don't worry too much about the `what()` message. It's nice to have a message that a programmer stands a chance of figuring out, but you're very unlikely to be able to compose a relevant and user-comprehensible error message at the point an exception is thrown. Certainly, internationalization is beyond the scope of the exception class author. Peter Dimov makes an excellent argument that the proper use of a `what()` string is to serve as a key into a table of error message formatters. Now if only we could get standardized `what()` strings for exceptions thrown by the standard library...

5 Expose relevant information about the cause of the error in your exception class's public interface. A fixation on the `what()` message is likely to mean that you neglect to expose information someone might need in order to make a coherent message for users. For example, if your exception reports a numeric range error, it's important to have the actual numbers involved available as numbers in the exception class's public interface where error reporting code can do something intelligent with them. If you only expose a textual representation of those numbers in the `what()` string, you will make life very difficult for programmers who need to do something more (e.g. subtraction) with them than dumb output.

6 Make your exception class immune to double-destruction if possible. Unfortunately, several popular compilers occasionally cause exception objects to be destroyed twice. If you can arrange for that to be harmless (e.g. by zeroing deleted pointers) your code will be more robust.

What About Programmer Errors?

As a developer, if I have violated a precondition of a library I'm using, I don't want stack unwinding. What I want is a core dump or the equivalent - a way to inspect the state of the program at the exact point where the problem was detected. That usually means `assert()` or something like it.

Sometimes it is necessary to have resilient APIs which can stand up to nearly any kind of client abuse, but there is usually a significant cost to this approach. For example, it usually requires that each object used by a client be tracked so that it can be checked for validity. If you need that sort of protection, it can usually be provided as a layer on top of a simpler API. Beware half-measures, though. An API which promises resilience against some, but not all abuse is an invitation to disaster. Clients will begin to rely on the protection and their expectations will grow to cover unprotected parts of the interface.

Note for Windows developers: unfortunately, the native exception-handling used by most Windows compilers actually throws an exception when you use `assert()`. Actually, this is true of other programmer errors such as segmentation faults and divide-by-zero errors. One problem with this is that if you use JIT (Just In Time) debugging, there will be collateral exception-unwinding before the debugger comes up because `catch(...)` will catch these not-really-C++ exceptions. Fortunately, there is a simple but little-known workaround, which is to use the following incantation:

```
extern "C" void straight_to_debugger(
              unsigned int,
              EXCEPTION_POINTERS*) {
  throw;
}

extern "C" void
   (*old_translator)(unsigned,
                     EXCEPTION_POINTERS*)
   = _set_se_translator(
                   straight_to_debugger);
```

This technique doesn't work if the SEH is raised from within a catch block (or a function called from within a catch block), but it still eliminates the vast majority of JIT-masking problems.

How should I handle exceptions?

Often the best way to deal with exceptions is to not handle them at all. If you can let them pass through your code and allow destructors to handle cleanup, your code will be cleaner.

Avoid `catch(...)` when possible

Unfortunately, operating systems other than Windows also wind non-C++ "exceptions" (such as thread cancellation) into the C++ EH machinery, and there is sometimes no workaround corresponding to the `_set_se_translator` hack described above. The result is that `catch(...)` can have the effect of making some unexpected system notification at a point where recovery is impossible look just like a C++ exception thrown from a reasonable place, invalidating the usual safe assumptions that destructors and catch blocks have taken valid steps to ensure program invariants during unwinding.

I reluctantly concede this point to Hillel Y. Sims, after many long debates in the newsgroups: until all OSes are "fixed", if every exception were derived from `std::exception` and everyone substituted `catch(std::exception&)` for `catch(...)`, the world would be a better place.

Sometimes, `catch(...)`, is still the most appropriate pattern, in spite of bad interactions with OS/platform design choices. If you have no idea what kind of exception might be thrown and you really must stop unwinding it's probably still your best bet. One obvious place where this occurs is at language boundaries.

David Abrahams

References

The following paper is a good introduction to some of the issues of writing robust generic components:

D. Abrahams: "Exception Safety in Generic Components", originally published in M. Jazayeri, R. Loos, D. Musser (eds.): *Generic Programming, Proc. of a Dagstuhl Seminar, Lecture Notes on Computer Science*. Volume 1766

SINGLETON – the anti-pattern!

by Mark Radford

A pattern captures and documents good practice that has been learned by experience. Patterns are a relative newcomer to software development, yet have actually existed in spirit within that community for as long as software has been developed. The point is this: skilled software developers have always known that when solving problems, some solutions seemed to work – with the benefit of prior experience, some solutions just *felt* right. A Pattern captures a problem and a solution that works, but that's not all, for it is very rare to find a solution that works in all circumstances. When experienced software developers apply a solution, they do so as a result of their experience, taking into account the context in which the problem occurs as well as the *tradeoffs* accepted in adopting that solution. Therefore, a Pattern captures a problem in context, together with a solution and its tradeoffs.

Patterns came to the attention of software developers in the 1990s and have accumulated a healthy body of literature. The book *Design Patterns* [1] is the best known and the one responsible for getting the mainstream of the community interested. It is rather ironic and sad, that this very book is also responsible for one of the worst red herrings ever to mislead the software developers: Singleton!

According to *Design Patterns* the intent of SINGLETON is to:

Ensure a class has one instance, and provide a global point of access to it.

Unfortunately this is rather vague, and this in itself causes some difficulties in discussing SINGLETON, because it fails to take into account that a SINGLETON is only meaningful if it has state. In the absence of state, ensuring there is only ever one object of a particular class is meaningless, because each instance is the same as every other one.

I have implemented SINGLETON many times over the past several years, and now, I can't think of one case where the SINGLETON solution was actually a good solution to the problem it attempted to solve. It seems to me that there are some serious problems with the whole approach, because correspondence between problem and solution domain models, encapsulation, and the ability to perform initialisation, are all compromised. Further, it is now my belief that the *Design Patterns* examples of where deployment of SINGLETON is claimed to be a good approach, fail to stand up under scrutiny (but more about that shortly).

I have remarked that patterns capture good ways of solving problems. However, for every good way there are many bad ones and some of these can be found deployed several times in practice. One reason for the repeated deployment of a bad solution is that it appears to solve the problem, and usually this is for one (or, for that matter, both) of two reasons:

- The problem that needs solving has not been correctly identified, and this results in the deployed solution being a solution to the wrong problem
- The solution has been deployed because it really does solve the problem, but subject to a set of tradeoffs ranging in quality from less than optimal to downright unsuitable

Such a recurring solution – i.e. one that leads to a worse rather than better design context – is known as an *anti-pattern*. It is my belief that SINGLETON is not a pattern but an anti-pattern, and that its use causes *design damage*! In this article I will attempt to state my case. I will start by listing several reasons why I think SINGLETON is a bad idea, and finish by making some recommendations for alternative approaches.

Problems

According to *Design Patterns*, SINGLETON is a *design* pattern – this means it is either language independent, or at least applicable to several languages. I will detail what I think the critical problems with SINGLETON are – i.e. the reasons why I choose to use the strong term *design damage* – with this factor in mind.

Design Models

I think the best way to proceed here, is to start by going back to the basics of interface design, and in particular the questions of:

- How much knowledge should an object be able to assume of the outside world in which it will be used?
- How much responsibility should be captured within a single interface?

The answer to the first of these is simply this: an object's knowledge of the outside world should extend to what it is told via its interface. The purpose of an object is to provide certain functionality to its clients, and to this end an object should provide the minimum useful interface that makes this functionality accessible. This underpins modularity in a design. How an object is used in the outside world *beyond* its interface is something that it should – as a matter of design principle – not assume any knowledge of. Therefore it follows that an interface can't make any assumptions about how many objects that support that interface are needed, because that issue is resolvable only in the outside world.

The question of how much responsibility an interface should be charged with is somewhat less concrete. An interface should be cohesive to the point that it embodies one role in a design, but where the boundaries of a role definition lie is by no means hard and fast. Consider an interface supporting a simple FACTORY METHOD: we can see that beyond whatever functionality the interface provides, its role is extended to also serve up other objects with related roles. However, in the case of SINGLETON, the interface must not only serve up the single object, but must also – in addition to whatever design role it plays – promise to manage that object. Its role therefore extends to three responsibilities.

Object design affords the capability to preserve correspondence between the problem domain model, through the stages of modelling the solution domain, and down to the implementation code. In his *Design* [2] presentation Kevlin Henney uses the term *modelarity* as meaning "a measure of the correspondence between the components of the problem being modelled and those in its solution". This *modelarity* factor alone plays a large part in accounting for the effectiveness of objects in software design. A key feature of a well designed system is the harmony between modelarity and modularity.

The whole premise on which SINGLETON is based is that there can only ever be one object of a certain class. When a design is viewed from the perspective of preserving modelarity, it suddenly becomes apparent that this premise is far from sound! In arguing this particular case, I will go back to *Design Patterns*, and examine what *it* claims are good uses of SINGLETON. In the section titled "Motivation" *Design Patterns* makes the following statements:

- Although there can be many printers in a system, there should only be one print spooler
- A digital filter will only have one A/D converter
- An accounting system will be dedicated to serving one company

The above three assertions all have something in common: they describe cases of a client (the system that uses the print spooler, the digital filter and the company) needing and using the services of only one instance of a supplier (the spooler, A/D converter and accounting system, respectively). Now *Design Patterns* asserts that the spooler, A/D converter and accounting system are therefore candidates for being SINGLETON. This however is not the case and would compromise modelarity, because there is no inherent reason why these three types of service supplier can only have one instance – the *real* case is that only one instance is needed by the client, and the real problem is that of how the client should manage the one instance it needs. Forcing the supplier to only ever have one instance deprives the model of its opportunity to express the cardinality. The model suffers because the *wrong problem* has been solved!

There is another way in which the use of SINGLETON compromises the harmony between modelarity and modularity. Here the problem is more subtle because SINGLETON appears to underpin modularity by putting an interface and the management of its instance in one place, but this is actually an illusion. The problem domain is the source of and motivation for the model, but the management of instances is a facet of the design of a software system – it does not happen in the problem domain. Therefore, instance management is a concern in its own right that should be separated from others. It follows that, while it could be argued that putting an interface and the management of its instance in one place constitutes modularity, the argument that this is the wrong modularity is far more compelling!

Encapsulation

Encapsulation is fundamental to object oriented design. It is the principle by which concerns are compartmentalised, and boundaries are drawn around them. Specifically, encapsulation manifests itself in software design in the form of implementation detail being kept cordoned off and used only via a public interface. It is this principle – the encapsulated implementation being accessed via a well defined public interface – that underpins many of the benefits that good design brings with it: clear communication of intended usage, ease of testability etc.

Global variables have been known to be the enemy of encapsulation for some time. SINGLETONs have but one instance, and it penetrates the scopes in which it is used via a route other than the public interface, making it the operational equivalent of a global variable. Therefore, SINGLETONs have many of the same drawbacks as global variables, and it is unfortunate that their appearance in *Design Patterns* has lead to so many software developers failing to notice this. In both cases – i.e. SINGLETONs and global variables – it becomes difficult to specify the pre and post conditions for the client object's interface, because the workings of its implementation can be meddled with from outside, but without going through the interface. A consequence of the difficulties in specifying pre and post conditions is that unit tests become harder to specify.

Initialisation

Usually, at the start of a program you won't have the information needed for a SINGLETON's initialisation. Initialisation on first use is no good, because you won't know which path will be taken through the program until it is actually run. If there is only one instance of a SINGLETON, then it must be initialised only once, on or just before the first use of the (unique) object. However, the path through the code will not be known until run time, and so there is no way to know the point at which the SINGLETON instance must be initialised.

One attempt to get around this I have seen (I'm deliberately not using the word "solution") is to attempt to initialise on every possible control flow on which a reference to the SINGLETON is obtained, with an exception being raised if it is used prior to initialisation. Besides being plain ugly, this approach introduces a maintenance headache; specifically this means:

- If a new control flow is introduced into the program and a reference to the SINGLETON object is acquired on it, then it is also necessary to ensure the initialisation is executed on the new control flow.
- When the test suite is updated to take account of the new control flow, an additional test – i.e. a test that fails if acquiring a reference to the Singleton object raises an exception – will be needed to ensure initialisation has taken place

All this is not to say that initialising the SINGLETON's instance is impossible, but the number of necessary workarounds and overheads can easily be seen mounting up.

Recommendations

Having given reasons why the use of SINGLETON causes damage to software design, what recommendations can be made for alternative approaches? It seems logical to look at the drawbacks described above, and suggest approaches that do not suffer from the same drawbacks. I'll start by addressing the latter two drawbacks – i.e. initialisation difficulties and breach of encapsulation – and then assess the situation. Consider the following two approaches, when a `Client` object uses the services of a `Supplier` object – `Supplier` being the object that would have been a SINGLETON, had such a design approach been used.

Approach	**When it makes sense**
Pass the `Supplier` object directly into the `Client`'s methods that make use of it, by passing the `Supplier` object directly through the interfaces of those methods.	When the `Client` object is not the sole user of the `Supplier` object
Pass the information (i.e. the arguments) needed to create the `Supplier` object to `Client` through its interface, so that `Client` can create the `Supplier` object within its implementation.	When the `Client` object is the sole user of the `Supplier` object

Both these approaches are examples of the pattern PARAMETERISE FROM ABOVE (see [3]). Actually, there's something familiar about these two approaches, and so there should be, because they're just describing normal design practices!

It is obvious that using either of the above approaches, there will be no problems initialising the `Supplier` object. In both cases `Supplier` can be initialised when it is created, be it in `Client` itself (latter approach) or in `Client`'s client.

Encapsulation is also significantly strengthened, and a good way to demonstrate this is to consider what happens when `Client`'s interface is unit tested. When the design approach used makes `Supplier` a SINGLETON the behaviour of `Supplier` is unpredictable because it is outside the control of both `Client` and *its* client. In this scenario the behaviour of the component under test – and hence the outcome of the test – is affected by something invisible and uncontrollable. Replace this scenario with one where either of the above two approaches is used and this changes as follows:

- In the former case `Supplier` can be replaced with a test implementation exhibiting behaviour designed to test `Client`
- In the latter case `Supplier` is an implementation detail of `Client`, so the lifetime of `Supplier` is encapsulated completely within `Client` and therefore there is no element of randomness about it

[concluded at foot of next page]

A Policy-Driven CORBA Template Library to Facilitate the Rapid Development of DOC Middleware

by Jeff Mirwaisi

While CORBA provides a robust, well-defined standard for the development of distributed object computing (DOC) middleware, the machinery needed to deploy a non-trivial application tends to be, at best, tedious and repetitive and, at worst, a source of hard-to-discover errors. As an application/system grows in complexity, the need for a reusable CORBA machinery abstraction becomes self-evident. The tried-and-true practice of cutting and pasting and the common setup function paradigm does not provide an adequate solution to the problem. In order to address these needs and overcome the problems of the classical design, this paper presents a policy-driven abstraction mechanism that promotes code reuse while preserving the rich capabilities CORBA provides.

1. Introduction

CORBA has developed into a robust, mature distributed object-computing standard over the years, but the effort in environment setup of large systems with many servant types is problematic. The cost of recreating the same or slightly differing requirements for multiple servant types or for multiple applications is unacceptable given that the reward for such work is non-existent and the power of modern development tools renders classical methods [1] inefficient. Large-scale CORBA applications and systems may provide hundreds of disparate servant types with varying requirements on the underlying ORB facilities. To meet the needs of CORBA server/system developers, the following presents a C++ policy-driven library design that eliminates the code redundancy issues in traditional CORBA applications while maintaining the rich set of options provided by the CORBA standard.

Template meta-programming has emerged as a powerful tool with which to construct reusable, extensible libraries. The CORBA Template Library (CTL) relies heavily on said techniques and borrows the policy-driven design approach pioneered by Andrei Alexandrescu in "Modern C++ Design" (MCD) [2]. The generic functionality this provides is ideal for library designers as it allows an extensibility lacking in traditional OO- and procedure-based libraries. Template-based policies provide the necessary paradigm to accomplish the "write once use many" goal, while providing a method for future extension that is both safe and predictable, and allowing us to leverage working code to dramatically cut down on testing, development, and redesign time.

2. Motivation

Expertise in the application of CORBA and the design of distributed computing systems is secondary to that of constructing a system that "does something." Work involved in the design and development of the machinery needed to deploy an application using CORBA is wasted effort because the real objective is the functionality exposed via CORBA. For instance, in order to set up a minimal CORBA server environment, an ORB must be initialized, a servant must be instantiated and activated, and an active thread must run the ORB dispatching mechanism. In the simplest case, this is not a burden and does not indicate a need for yet another library/abstraction. A transient servant could simply be activated under the Root POA and the application's main thread would run the dispatch loop. If, however, we were to need a system with many servant instances, multiple POAs with differing POA policies and ORB requirements, the task becomes substantially more difficult. If we wish to develop a true peer-to-peer application, where the application is both a client and a server, instead of a pure server application, the necessary setup becomes even more cumbersome.

Historically, these concerns have been met with procedural setup and initialization functions, code repetition, or an OO abstraction mechanism, none of which are ideal. Procedural solutions fall short because of lack of easy reconfiguration of servant initialization behavior. Environmental changes require modification of servant setup code, possibly in multiple locations, for instance, calls to a

[continued from previous page]

Now that initialisation and encapsulation are taken care of, what of the issues related to models and class design? Well, interface design seems to be in good shape: what could be more natural than passing an object – either a supplier object, or the information needed to create one – through an object's interface? Unfortunately, the approaches recommended here will not automatically avoid compromising any models. What *is* for certain is that no models are automatically compromised either, which would be the case were an approach involving SINGLETON to be used (for reasons described earlier in this article). It was SINGLETON's mixing of concerns in its interface – its role in the model and the object management concern – that was problematic, but provided participating interfaces are designed with attention being paid to cohesion, this does not happen when a PARAMETERISE FROM ABOVE approach is adopted.

Summary

For a problem and its solution to be a pattern, the solution must be a good one *in practice*. SINGLETON is based on the premise that a class must only ever have one instance, and must itself enforce this singularity – but the premise is false because the client of the class, not the class itself, is in a position to know how many instances are needed. Further, breaching encapsulation and causing initialisation difficulties cannot be good for any set of design tradeoffs. Given the design damage that SINGLETON inflicts, it must be considered an *anti-pattern.*

Mark Radford
mark@twonine.co.uk

References

[1] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.
[2] Kevlin Henney, *Design: Concepts and Practices*, Keynote presentation at JaCC, Oxford, 18th September 1999 (available from www.curbralan.com).
[3] PARAMETERISE FROM ABOVE is a term in use, but there is currently no formal write-up.

Acknowledgements

Thanks to Kevlin Henney and Alan Griffiths for their helpful comments.

number of different POA setup procedures and object instantiation and activation functions. A simple change from a transient, system-identified, stack-allocated servant to a persistent, user-identified, heap-allocated servant is error prone and takes more effort than the alternate policy-driven model presented here. The traditional OO-based design also fails to simplify deployment significantly and can in fact be more difficult to adapt than the procedural solution. Inheritance-based solutions rely on a given base type that limits your design choices or requires a large number of similar base types to achieve all possible combinations, i.e., a reference counted transient servant, a non reference counted version, and persistent version of both, etc.

Addressing these limitations with a reusable and extensible template library provides an abstraction layer that will allow developers to concentrate on the functionality they want to expose, not the machinery needed to expose them. To that end, the following presents some meta-programming techniques to simplify our goal, and a policy driven library of reusable types that simplify DOC middleware development to a point where the only new work needed in most cases is the development of the methods exposed by a given interface. The following pages present the basic techniques employed by the library and a synopsis of the library policies and their usage.

3. CORBA Template Library (CTL)

The CORBA Template Library (CTL) presented here attempts to provide an extensible library of policies and composable elements to simplify the deployment of large-scale CORBA applications. The design concentrates on the servant activation process, the mechanism by which the servant accesses the information it needs during the activation process, and the organization of servant instances in a larger service framework. The CTL is a freely available C++ source code library, available under the BSD license.

Currently the library is tied to ACE/TAO but work is underway to make it as ORB agnostic as possible.

3.1 Servant Activation/Deactivation

Servants represent the concrete implementation of a CORBA Object in a particular programming language. They are local instances of a type that are made known to the CORBA framework in order to service remote requests. A local servant instance is “activated,” made known to the CORBA framework, by first instantiating the servant, finding or creating a suitable Portable Object Adapter (POA), registering with that POA, and finally some secondary object-activation activity such as Object exposition through the Naming Service. In order to preserve the rich set of options available to CORBA developers a policy-based design was chosen that allows every set of possible activation scenarios to be expressed by way of four policy-type delegates: memory policies, POA- activation/selection policies, object-activation policies, and auxiliary policies. The servant activation (`createServant()`) and deactivation (`destroyServant()`) functions simply dispatch the requests to the specified policy delegates.

3.1.1 Memory Activation/Deactivation

The memory policy is the first called during activation and the last called during deactivation. Most users will simply use the empty memory policy, which does nothing. But a number of

Basic CORBA Terminology

CORBA, Common Object Request Broker Architecture: CORBA is a standardized middle-ware framework that facilitates distributed-object system development. The CORBA standard is maintained and developed by the OMG (Open Messaging Group, `www.omg.org`) Among other things the standard provides specific mappings between a platform independent “interface definition language” (IDL) and a number of development languages such as C++ and Java, a transparent messaging infrastructure for marshalling requests for, and data to, the remote processes, and a set of well defined Services such as the Naming Service.

ORB, Object Request Broker: the ORB is the messaging component of a CORBA system, all outgoing requests go through an ORB and are packaged and marshaled across the wire, all incoming requests are de-marshaled and dispatched by the ORB. The ORB has a number of secondary functions as well such as bootstrapping (resolve_initial_references), IOR management/manipulation (string_to_object, object_to_string) etc.

POA, Portable Object Adapter: the POA is the bridge between the ORB and a servant skeleton. ORBs dispatch a remote request/invocation to a POA that in turn either rejects the message or delivers it to a concrete servant implementation by way of its skeleton.

CORBA Object: Objects are the CORBA counterparts to servant instances, while the servant is a process local entity, an Object is accessible across process and machine boundaries.

IOR, Interoperable Object Reference: IORs are encoded names that uniquely identify an Object, they include the host’s contact information, the object key that identifies the Object to the ORB/POA, and any other data necessary to contact the Object.

IDL, Interface Definition Language: IDL is a platform independent language to describe interfaces and data types. The IDL is run through an IDL compiler that generates language specific mappings of the types. Skeleton and stub code is generated by the IDL compiler.

RootPOA: The Root POA is the primary POA hosted by an ORB; all other POAs are subordinates of the RootPOA. Simple transient Objects with system ids can be activated under the RootPOA. It is also the only POA immediately available when the ORB is initialized and the last to be destroyed during deactivation.

Name Service: the Naming Service (CosNaming) is a CORBA service for exposing objects in a directory like manner – advertised object references are identified by a name and kind and placed in a naming-context, which is similar to a directory or folder. Clients can resolve names (files) and contexts (directories) bound to the Name Service obtaining an Object reference for the associated name that can be narrowed to an expected interface.

Skeleton: generated server side code that bridges the gap between the POA and the servant, conforms to an expected interface used by the POA and provides the mapped IDL types interface to be overridden by the servant implementation

Stub: generated client side code that supports the IDL specified interface and interacts with the ORB to reach a remote Object, a remote proxy to the Object and servant.

Servant: user generated code that conforms to the interface expected by a skeleton and implements the necessary code to fulfill the methods of the IDL specified interface.

policies are provided to simplify garbage collection of free store allocated servants. The memory policy also provides a convenient base type for alternate allocation strategies, such as a pooled allocator.

3.1.2 POA Activation/Deactivation

The POA activation process is arguably the most cumbersome aspect of CORBA applications, and generates the most redundant code. While it is possible to use a procedural approach that simplifies the most common-use scenarios, with seven basic policies and a number of extension policies (pluggable protocols such as bi-directional GIOP) an exhaustive set of POA activation procedures is not a practical option.

The CTL provides two mechanisms by which a POA can be created: two functions `create_poa` and `create_from_poa`, which are explicitly specialized with the policy type desired; and a servant POA policy that will create the POA for a servant during the activation process. The two options allow for easy POA activation whether the POA to Object relationship is one-to-many or one-to-one.

A classic POA activation scenario involves first obtaining and narrowing a reference to the `RootPOA` obtaining the `POAManager`, generating the policy list, creating the new POA, and then cleaning up.

```
CORBA::Object_var obj
      = orb->resolve_initial_references("RootPOA");
PortableServer::POA_var root
      = PortableServer::POA::_narrow(obj);
PortableServer::POAManager_var poa_manager
      = root->the_POAManager();
CORBA::PolicyList policies;
policies.length(2);
policies[0] = root->create_lifespan_policy(
                       PortableServer::PERSISTENT);
policies[1] = root->create_id_assignment_policy(
                       PortableServer::USER_ID);
PortableServer::POA_var wfa_poa = root->create_POA(
                       "WidgetFactoryAdmin_i",
                       poa_manager,policies);
poa_manager->activate();
for(CORBA::ULong i=0;i<policies.length();++i)
  policies[i]->destroy();
```

The equivalent when using CTL is to specify a POA activation policy during servant activation:

```
POAPolicy::FromRoot<D,POAPolicy::PersistentUserID<> >
// POA policies which require an id to create the
// POA also provide a set_poa_id method.
```

And an example when using CTL to create a new POA outside of servant activation:

```
PortableServer::POA_var w_poa = POAPolicy::create_poa<
   POAPolicy::PersistentUserID<> >(orb,"Widget_i");
```

The POA policy types provided are composable in a linear hierarchy, as are most of the types in the CTL. So for instance, assuming you wanted a POA that used bi-directional GIOP, hosted transient system id Objects, and didn't allow implicit activation of servants, the appropriate specification would merely be a string of nested template types:

```
POAPolicy::Bidirectional<
          POAPolicy::TransientSystemID<
                    POAPolicy::NoImplicit<> > >
```

POA deactivation, in both the classic and CTL use scenarios, is relatively simple. The CTL POA policy takes care of it automatically if it created the POA. If the POA was not created during the activation process but was instead created as above the user is responsible for destroying the POA when it is no longer needed, for instance:

```
w_poa->destroy(true,true);
```

3.1.3 Object Activation/Deactivation

Once a suitable POA has been created a servant instance must be tied to the POA and an Object reference that specifies the servant to the outside world must be generated. A persistent Object reference remains the same across different execution contexts while a transient Object reference is unique each time the servant is activated. CTL provides two simple policy types that activate persistent and transient Objects under the specified POA during the activation process. A number of other policies are also provided such as Multicast servant activation but are not discussed here.

A classic persistent Object activation scenario:

```
PortableServer::ObjectId_var oid
     = PortableServer::string_to_ObjectId(
                          "WidgetFactoryAdmin_i");
wfa_poa->activate_object_with_id(oid.in(),&wfa);
CORBA::Object_var wfa_obj
     = wfa_poa->id_to_reference((oid.in()));
Example::WidgetFactoryAdmin_var wfa_ref
     = Example::WidgetFactoryAdmin::_narrow(wfa_obj);
```

In contrast, when using the CTL, activating a servant with a persistent Object reference is simplified to specifying the object-activation policy:

```
ObjPolicy::UserID<D>
// object policies which require an id during
// activation expose a set_object_id method.
```

Object deactivation is done automatically during servant deactivation in the case of CTL servants. A classic CORBA application would need to explicitly deactivate the Object when it is no longer needed:

```
PortableServer::ObjectId_var oid
      = wfa_poa->servant_to_id(&wfa);
wfa_poa->deactivate_object(oid);
```

3.1.4 Auxiliary Activation/Deactivation

Auxiliary policy processing is the last step during activation and the first during deactivation. It provides a point during the activation/deactivation process after the servant has been associated with a CORBA Object and before, in the case of deactivation, that Object has been torn down. As such it is an ideal point from which to expose the Object to some form of external lookup such as the Naming Service; or to set up messaging policies to be used, such as timeout policies. Two simple exposition policies are presented here: one that binds the newly activated Object to the ORB's IOR table, and one that binds the Object to the root naming-context of the Naming Service.

The IOR table provides a mechanism by which external ORB processes can discover an Object and obtain a reference to the Object. When a call to `orb->resolve_initial _references("NameService")` is made, if an initial reference was not supplied during ORB initialization a multicast request is sent to external ORB processes that in turn query their IOR table to try to fulfill the request for the name specified(`"NameService"` in this case). A CTL servant can request IOR table binding and unbinding during activation and deactivation respectively, by providing the appropriate Auxiliary policy:

```
AUXPolicy::BindIORTable<D>
// The BindIORTable policy exposes a method
// set_iortable_id to specify the ID the Object
// reference will be associated with in the table.
```

The equivalent classic CORBA application would need to:

```
obj = orb->resolve_initial_references("IORTable");
IORTable::Table_var tbl
      = IORTable::Table::_narrow(obj);
CORBA::String_var str
      = orb1->object_to_string(wfa_obj);
tbl->rebind("WidgetFactoryAdmin_i",str.in());
```

in order to bind the Object reference to the IOR table. And in order to unbind do the complementary

```
tbl->unbind("WidgetFactoryAdmin_i");
```

during deactivation. This of course entails either maintaining the above `IORTable::Table_var` or reacquiring it when needed. The corresponding CTL use is transparent and automatically done during the servant deactivation process.

The Naming Service provides a directory service for exposing “well known” servants to distributed systems. Much like the IOR Table it provides a mechanism for exposition and lookup. CTL servants that wish to take advantage of the Naming Service simply specify the appropriate auxiliary policy such as:

```
AUXPolicy::BindRootNamingContext<D>
// The BindRootNamingContext policy provides a
// set_cosnaming_data method to specify the id and
// kind that will be bound to the root naming-context.
```

to be used during activation and deactivation. A classic CORBA application on the other hand needs to manually register with the Naming Service after the Object has been activated:

```
CosNaming::Name name;
name.length(1);
name[0].id
      = CORBA::string_dup("WidgetFactoryAdmin_i");
name[0].kind
      = CORBA::string_dup("WidgetFactoryAdmin_i");
obj = orb->resolve_initial_references("NameService");
CosNaming::NamingContext_var nc
      = CosNaming::NamingContext::_narrow(obj);
nc->rebind(name,wfa_obj);
```

And correspondingly unbind before deactivation:

```
nc->unbind(name);
```

Like the other CTL policy types the Auxiliary policies may be combined into a composite. So if a servant wanted to bind both to the IOR table and the Naming Service a composite policy type:

```
AUXPolicy<BindIORTable<D,
                 AUXPolicy::BindRootNamingContext<D> >
```

could be used.

3.2 Mix-in Composition: Host and Delegate

The CTL makes extensive use of the Curiously Recurring Template Pattern and Parameterized Inheritance to deliver a “composable” design. Composable in this context refers to the ability to extend an inheritance hierarchy by nesting template types. The main use for this technique in the CTL is to provide a consistent means by which to retrieve the necessary data needed during servant activation/deactivation. The choice of whether to “host” the data in the servant type or to “delegate” the request to some other type is left up to the user. Hosts store the data as a member of the servant type and provide a common interface used by the policies to retrieve that data. Delegates provide the same interface but act as proxies for the data. The CTL provides a number of Host and Delegate mix-in types in the `CTL::MIX::Host` and `CTL::MIX::Delegate` namespaces respectively.

All the types in the MIX namespace use both a `D` (Derived) and `B` (Base) template parameter and are therefore “composable” as nested templates. Assuming `S` is the servant base type and we wished to construct a servant that hosts its own `ORB_var` (`CTL::MIX::Host::ORB`) and `POA_var` (`CTL::MIX::Host::POA`) we could form the composite type by writing out the template `CTL::MIX::Host::ORB<D, CTL::MIX::Host::POA<D, S> >` where `D`, as noted above, represents the most derived type i.e. the type that inherits from the composite. To simplify the above, which gets rather ugly when many types are composed, the `CTL::Compose` namespace provides a family of templates that will allow the type to be expressed in a simpler manner. For instance, the `composeDB` type could be use to rewrite the above as `CTL::Compose::composeDB<D,S, CTL::MIX::Host::ORB, CTL::MIX::Host::POA>::type`, or without the fully qualified names `composeDB<D,S,ORB, POA>::type`.

For a further discussion of composition see the sidebar Template Techniques Used in the CTL (next page), and the material in the references section. The following sections present some of the more important elements of the `CTL::MIX` namespace.

3.2.1 Mix-in ORB

Every servant needs a reference to an ORB so that a POA can either be found or created, and an Object can be activated. The ORB mix-in type provides an interface for accessing the `ORB_var` that the servant type will use. The Host variation maintains a member `ORB_var` and provides an additional method `set_orb`, while the Delegate dispatches the request to the servant’s “container”.

3.2.2 Mix-in ORBTask

CTL provides a helper type `CTL::OrbTask` that is simply an `ORB_var` and thread that runs the ORB reactor loop. The mix-in type `CTL::MIX::Host::ORBTask` helps simplify many multi-orb applications.

3.2.3 Mix-in POA

Once a POA has been created or found on behalf of a servant, it needs to be stored for subsequent use during Object activation and deactivation. The mix-in types `CTL::MIX::Host::POA` and `CTL::MIX::Delegate::POA` provide an interface that returns a `PortableServer::POA_var&` that the activation/deactivation policy methods can use to store and gain access to the associated POA. The Host variety also provides a `set_poa` method.

3.3 Servant Holders

Once a servant has been activated and an Object reference (IOR) has been generated that data needs to be stored somewhere. The CTL provides two simple holder types, `CTL::BasicServant` and `CTL::TunneledServant`. The only difference between the two is that `CTL::TunneledServant` allows the storage of a second IOR that identifies the servant, for instance an alternate route through a DSI gateway. Both Holder types support a `getServantPtr` and `setServantPtr` method. `CTL::TunneledServant` supports the additional methods `getTServantPtr` and `setTServantPtr`.

3.4 Container Holders

An optional Holder type that maintains a parameterized-back-pointer to some arbitrary type is also provided. `CTL::BasicContainer` maintains this data and provides a simple interface for modifying and accessing the information. The Delegate types in `CTL::MIX::Delegate` use this information to redirect requests for activation data to the type specified to `CTL::BasicContainer`. The interface supported is very simple: `getContainer`, `attachToContainer` and `detachFromContainer`.

3.5 Servant State

`CTL::ServantState` acts as a common base that servants can use to report activation and deactivation failure. The interface is simple and merely provides a method of setting and querying a bit field that represents different servant states. All CTL servants support the `CTL::ServantState` interface.

3.6 A Complete Servant

All CTL servants provide a simple interface: `createServant()` and `destroyServant()`. These in turn use the policy types provided to activate and deactivate the servant. Though the interface is simple the templated nature of the policy driven design allows for an unlimited set of servant activation/deactivation scenarios. Two base servant types are presented below. They are identical except that the second inherits from an extra "container" template type that represents a container holder type as described in section 3.4. The number of template parameters may seem extravagant but they allow us to formulate a generic servant activation/deactivation mechanism.

```
template <typename  D,typename S, typename SS,
          typename PP = CTL::POAPolicy::Empty,
          typename OP = CTL::ObjPolicy::Empty,
          typename MP = CTL::MemPolicy::Empty,
          typename AP = CTL::AUXPolicy::Empty,
          typename B = CTL::EmptyType >
struct CompositionServant : CTL::CompositionType<D,B>,
               S,SS,MP,PP,OP,AP,CTL::ServantState {
  typedef B BASE;
  typedef D DERIVED;
  virtual void createServant() {
    initializeMemoryPolicy(
                static_cast<DERIVED*>(this));
    initializePOAPolicy(
                static_cast<DERIVED*>(this));
    initializeObjectPolicy(
                static_cast<DERIVED*>(this));
    initializeAUXPolicy(
                static_cast<DERIVED*>(this));
  }
  virtual void destroyServant() {
    destroyAUXPolicy(static_cast<DERIVED*>(this));
    destroyObjectPolicy(
                     static_cast<DERIVED*>(this));
    destroyPOAPolicy(static_cast<DERIVED*>(this));
    destroyMemoryPolicy(
                     static_cast<DERIVED*>(this));
  }
  virtual ~CompositionServant() {}
};
```

And the similar servant base type that also provides a template parameter for a container holder type.

Template Techniques used in the CTL

Fulfilling the requirements of an extensible, reusable library requires a number of template techniques. A basic understanding of template meta-programming is assumed, as is a familiarity with policy-driven design as expressed in "Modern C++ Design" (MCD) [2]. Beyond basic template techniques and the concepts propounded in MCD, a number of other mechanisms are used. The major techniques used in CTL are briefly described below.

Policy-based Design

Though the concept of policy driven designs is not completely new (POA policies), the concept has been adopted in recent years by the generic programming community, thanks in large part to the pioneering work presented in "Modern C++ Design" (MCD) and the Loki library. Policies represent the behavior exposed by a type (P) when dealing with a particular type (T) and, therefore, are groups of related functions, in contrast to traits, which describe a particular type (T) by way of a set of descriptive typedefs and values.

There are a number of ways to represent policies, the simplest of which is a class template that is made available to the client type via a template template type. Unfortunately, template template types are just now being handled consistently across different C++ compiler implementations. Therefore, CTL uses the simpler form of a specialized templated struct specialized by the client, except in the situation where the policy is not inherited. In those cases the policy design is simplified and a template type, whose methods are static, is expected. A complete discussion of policy-based meta-programming is not given here, but for the sake of completeness the following gives a quick demonstration.

A simple example:

```
template <typename T>
struct P {void f() {}};
```

`P` is a very simple policy for an arbitrary type `T`. In the above example, `P` makes no demands on the type `T`. This is of course not the general case but is merely a simplification used to illustrate the policy paradigm.

```
template <template <typename U> class T>
struct C : T<C<T> > {};
```

C is an arbitrary client of policies. It designates that one arbitrary policy is needed (T) and will inherit from said policy, specifying itself as the parameter type.

```
typedef C<P> CP;
```

CP is a completely defined type, which uses the P policy above in conjunction with the C policy client.

A number of resources dealing with policies [2] and templates [4] are provided in the references.

The Curiously Recurring Template Pattern

The Curiously Recurring Template Pattern (CRTP) [5], for lack of a better term, is a template technique that introduces the most derived type as a parameter known to the base type at compile time. This fact allows the base type to make up-calls via a static cast of the instance to the derived type's complete interface. Effectively, this imposes interface requirements on the derived type. This is similar to an abstract interface, which needs to be overridden in the concrete type, without the necessity for a virtual function and the extra baggage of a formal abstract interface. Templates allow us to extend the interface concept by breaking away from the large-scale (whole interface) contract, to a method-by-method contract. The base type does not require that the derived type inherit from a particular base, only that it provides the methods we need.

A simplified example:

```
template <typename D>
struct U {
  void f() {static_cast<D*>(this)->g();}
};
```

U represents a template base type. It requires that derived types implement some function: `<arbitrary return type> g()`. Obviously U can require any signature for g. The simplest possible signature is required in the above example.

```
struct V : U<V> {void g() {}};
```

V represents a complete hierarchy. It implements the method as required by the base type (U) and inherits from a specialization of U, designating itself as the most derived type.

Parameterized Inheritance

Parameterized inheritance allows us another degree of freedom when creating a type. While traditional inheritance allows us to specify the access of the base type (public, protected, private), the number of copies of the base to keep (virtual), as well as whether a single or multiple base type(s) are the ancestor of the type, parameterized inheritance leverages the template system to inherit from an arbitrary type on demand. This allows us to reuse the same method implementation without regard to base type, thereby producing a true mix-in on-demand development strategy, which greatly enhances our ability to reuse code.

A simplified example:

```
template <typename B>
struct M : B {virtual void f() {}};
```

M represents a parameterized inheritance mix-in type, which provide an implementation of a method f() that can be used to make disparate abstract types concrete (assuming they only require an override of a pure method `void f()`).

```
struct U {virtual void f()=0;};
struct V {virtual void f()=0;};
```

U and V represent two arbitrary unrelated abstract types with a common method (`void f()`). Derived types "could" either reproduce the common implementation in the concrete types, leading to code redundancy, or delegate to a non-member function common to both. Among other issues, this requires us to provide a delegating implementation in both concrete types (code redundancy) and, unless the common method is templated, either does not allow us to access state information encapsulated in the instance or forces us to inherit from a fixed base and use a polymorphic call from the common function being delegated to. Note that when combined with the above technique (CRTP), M's implementation of f() can use state members and methods of the complete type.

```
struct CU : M<U> {};
struct CV : M<V> {};
```

CU and CV are concrete implementations of U and V, respectively. They are complete as a consequence of the shared implementation of f() implemented in M.

Parameterized Back-Pointer

Parameterized Back-Pointer formalizes a common methodology and abstracts it by using the generic meta-programming power of C++ templates. It is common practice to expect a member type to use a reference to the containing type.

```
struct S {
  struct T {
    T(S* s) : s_(s) {}
    S* s_;
  };
  S() : t_(this) {}
  T t_;
};
```

A similar scenario arises when we simply want a type to have a reference to an unrelated type that exposes an expected interface.

A simple example:

```
template <typename C>
struct CT {
  C* c_;
  CT(C* c) : c_(c) {}
  void g() {c_->f();}
};
```

CT represents a simple type that uses a parameterized back-pointer. It expects a reference to some arbitrary type (C) that has a matching signature for f(). The relation between the parameter C and CT is not necessarily one of containment (i.e., CT is a member of C) but we use the term containment in lieu of a better term.

```
struct U {
  void f() {}
  CT<U> ct_; U() : ct_(this) {}
};
```

U is an arbitrary type that supports the method f() as expected by CT, and contains an instance of CT (ct_) that is constructed with a pointer to the current U instance.

```
template <typename  D, typename S, typename SS,
          typename PP = CTL::POAPolicy::Empty,
          typename OP = CTL::ObjPolicy::Empty,
          typename MP = CTL::MemPolicy::Empty,
          typename AP = CTL::AUXPolicy::Empty,
          typename C = CTL::EmptyType,
          typename B = CTL::EmptyType >
struct ContainedCompositionServant
      : CTL::CompositionServant<
                          D,S,SS,MP,PP,OP,AP,B>, C {
  typedef B BASE;
  typedef D DERIVED;
  virtual ~ContainedCompositionServant() {}
};
// The D parameter represents the Derived type.
// The S parameter represents a Servant holder type
//      i.e. BasicServant<Example::Widget>
// The SS parameter represents the skeleton type
//      i.e. POA_Example::Widget
// PP is the POA policy
// OP is the object-activation policy
// MP is the memory policy
// AP is the Auxiliary policy
// B is an arbitrary base type
// C, which is used in the second type, is the
//      Container Holder type.
```

Note that the resultant types inherit from all the template parameters and have their combined interface.

As shown above the `CTL::ContainedCompositionServant` type reuses the `CTL::CompositionServant` type. The `CTL::CompositionType` is not discussed here, but can be considered as `B` (the arbitrary base parameter).

To illustrate that the above is not as complicated as it seems from initial inspection, let's consider a simple servant. The servant will have a transient system id POA, and register with the Naming Service when activated.

```
module Example {
  interface MyType { string fn(); }
};
```

The types `Example::MyType` and `POA_Example::MyType` are generated from the above IDL. Below is a ready to use servant type and server. Note that the fully qualified names are dropped for readability.

```
struct simple : composeDB<
    simple, CompositionServant<
          simple,
          BasicServant<Example::MyType>,
          POA_Example::MyType, POAPolicy::FromRoot<
                POAPolicy::TransientSystemID<> >,
          ObjPolicy::SystemID<>, MemPolicy::Empty,
          AUXPolicy::BindRootNamingContext<
                simple> >,
    Host::ORB, Host::POA >::type {
  simple() {
    set_cosnaming_data("simple", "simple kind");
  }
  char* fn() {
    return CORBA::string_dup("Hello World");
  }
};
```

```
int main(int argc,char* argv[]) {
  simple s;
  s.orb_ = CORBA::ORB_init(argc,argv);
  s.createServant();
  s.orb_->run();
  return 0;
}
```

The above is a complete server application, `simple::createServant()` will construct the POA, activate the servant and register it with the Naming Service using the id `"simple"` and the kind `"simple kind"`. If a client resolved the Object reference in the Naming Service and invoked the `fn()` method, the string `"Hello World"` would be returned.

4 A Widget Factory Example

To illustrate the power and ease of use of the CTL, the following presents an example that uses many of the CTL features. Experienced CORBA developers will note how much simpler the code is than a similar traditional application [1]. An equivalent classic application can be found on the ACCU website. Beyond the exotic-looking base types, the code is almost reduced to that of a standard C++ application. Furthermore, if an error exists in the setup machinery, we have a single point of failure and only a single policy that needs to be debugged.

The example demonstrates a simple Widget server application. The server uses two ORBs, one for public access to the factory's interface and one that provides private access (from the local machine) to the factory's administrative interface. The two different servants that implement these interfaces share common information and are in fact part of a larger Widget Factory "component". Both the servants of the factory component are persistent Objects and each advertises its existence with the Naming Service and IOR table. The publicly accessible interface provides a method for Widget creation while the private admin interface provides a method to destroy all the Widgets created and a method to shut down the Widget Factory.

```
<example.idl>
module Example {
  interface Widget {
    void do_something();
  };
  interface WidgetFactory {
    Widget create_widget();
  };
  interface WidgetFactoryAdmin {
    void destroy_all_widgets();
    void shutdown_widget_factory();
  };
```

The above IDL describes the three basic interfaces and types that the following Widget factory will implement. Once it is run through an IDL compiler a skeleton and stub will be generated for `Widget`, `WidgetFactory`, and `WidgetFactoryAdmin`.

The servants presented here make use of a simple helper type `CTL::RefCountImpl` that extends the functionality of `PortableServer::RefCountServantBase` by providing methods to get the current reference count and to wait on a specific count.

```
template <typename D,typename S,
          typename SS,typename C>
```

```
struct factory_servant
     : ContainedCompositionServant<
            D,BasicServant<S>,
            SS, MemPolicy::Empty,
            POAPolicy::FromRoot<
                 D,POAPolicy::PersistantUserID<> >,
            ObjPolicy::UserID<D>,
            AUXPolicy::BindRootNamingContext<
                 D,AUXPolicy::BindIORTable<D> >,
            BasicContainer<C*>,
            RefCountImpl<
                 D,PortableServer
                       ::RefCountServantBase> > {};

template <typename D,typename S,
          typename SS,typename C>
struct simple_servant
     : ContainedCompositionServant<
            D,BasicServant<S>,
            SS, MemPolicy::Empty,
            POAPolicy::Empty,
            ObjPolicy::SystemID<D>,
            AUXPolicy::Empty,
            BasicContainer<C*>,
            RefCountImpl<
                 D,PortableServer
                       ::RefCountServantBase> > {};
```

These two servant base types will allow us to reuse the groups of policies without restating them for each servant type. The `factory_servant` base type will be shared between the `WidgetFactory_i` and `WidgetFactoryAdmin_i` types. The `Widget_i` type will use the base type `simple_servant`.

```
template <typename C>
struct Widget_I
     : composeDB<Widget_i<C>,
           simple_servant<Widget_i<C>,
               Example::Widget,
               POA_Example::Widget,C>,
           Delegate::ORB,Delegate::POA>::type {
  void do_something() throw (CORBA::Exception){}
};
```

Widgets are the product created by our factory for use by remote clients. Once `Widget_i` has been specialized for the container type it is a ready-to-use servant type. When `createServant()` is called the corresponding CORBA Object is activated and an Object reference is generated. `WidgetFactoryAdmin_i` will eventually call `deactivateServant()` for each of the `Widget_i` servants created by the factory.

```
template <typename C>
struct WidgetFactory_I
     : composeDB<WidgetFactory_i<C>,
           factory_servant<WidgetFactory_i<C>,
               Example::WidgetFactory,
               POA_Example::WidgetFactory, C>,
           Host::ORBTask,
           Host::POA >::type {
  WidgetFactory_i() {
    set_cosnaming_data("WidgetFactory_i",
                       "WidgetFactory_i");
    set_object_id("WidgetFactory_i");
    set_poa_id("WidgetFactory_i");
    set_iortable_id("WidgetFactory_i");
  }
  Example::Widget_ptr create_widget()
              throw (CORBA::SystemException) {
    ACE_Guard<ACE_Thread_Mutex>
                       grd(getContainer()->mtx_);
    Widget_i<C> * p = new Widget_i<C>;
                  //exception unsafe for clarity
    p->attachToContainer(getContainer());
    p->createServant();
    getContainer()->widget_data_.push_back(p);
    return p->getServantPtr();
  }
};
```

The `WidgetFactory` interface is the publicly visible portion of our component; any remote client can request a new `Widget`. `WidgetFactory_i`, a servant type that supports the `WidgetFactory` interface, is a template that generates a servant once it is specialized with a container type. The `C` (container) parameter will later be provided as `WidgetFactoryData`. Before the servant is activated, the user will provide a `(C*) WidgetFactoryData*` by way of `attachToContainer`, allowing us to gain access to the vector of `Widget_i<WidgetFactoryData>*` that is shared between the public factory interface's servant and the private factory admin interface's servant that is a member of `WidgetFactoryData`.

```
template <typename C>
struct WidgetFactoryAdmin_I
     : composeDB<WidgetFactoryAdmin_i<C>,
           factory_servant<
                WidgetFactoryAdmin_i<C>,
                Example::WidgetFactoryAdmin,
                POA_Example::WidgetFactoryAdmin,C>,
           Host::ORBTask,
           Host::POA >::type {
  WidgetFactoryAdmin_i() {
    set_cosnaming_data("WidgetFactoryAdmin_i",
                       "WidgetFactoryAdmin_i");
    set_object_id("WidgetFactoryAdmin_i");
    set_poa_id("WidgetFactoryAdmin_i");
    set_iortable_id("WidgetFactoryAdmin_i");
  }

  void destroy_all_widgets()
              throw (CORBA::SystemException) {
    ACE_Guard<ACE_Thread_Mutex>
              grd(getContainer()->mtx_);
    for(std::size_t i=0;
        i<getContainer()->widget_data_.size();
        ++i) {
      //simple loop for clarity
      getContainer()
            ->widget_data_[i]->destroyServant();
      getContainer()
            ->widget_data_[i]->_remove_ref();
    }
    getContainer()->widget_data_.clear();
  }
```

```
    void shutdown_widget_factory()
                throw (CORBA::SystemException) {
      getContainer()->event_.signal();
    }
  };
```

WidgetFactoryAdmin_i is similar to the above WidgetFactory_i, except for the interface the servant will expose to remote clients. It also requires a template parameter to designate its “container” type, which will later be specified as WidgetFactoryData. Like the above servant type it also hosts its own CTL::OrbTask and POA. Because WidgetFactory_i and WidgetFactoryAdmin_i operate on independent ORBs a request coming in on WidgetFactory_i’s ORB can not gain access to the WidgetFactoryAdmin_i. Therefore if the ORB hosting the admin servant is listening on localhost:10000 only local access is granted to the admin interface.

```
  struct WidgetFactoryData {
    ACE_Event event_;
    ACE_Thread_Mutex mtx_;
    PortableServer::POA_var widget_poa_;
    WidgetFactory_i<WidgetFactoryData> factory_;
    WidgetFactoryAdmin_i<WidgetFactoryData>
                               factory_admin_;
    std::vector<Widget_i<WidgetFactoryData>*>
                               widget_data_;
    // widget delegates to container to
    // determine orb to use
    CORBA::ORB_var& orb(Widget_i<
         WidgetFactoryData>*) {return factory_.orb_;}
    // widget delegates to container
    // to determine which POA to use
    PortableServer::POA_var& poa(Widget_i<
         WidgetFactoryData>*) {return widget_poa_;}
  };
```

WidgetFactoryData is a “container” for all three servant-types. It provides the necessary interface for the Widget_i servant delegates, and specializes the WidgetFactory_i and WidgetFactoryAdmin_i templates with itself as the container type. Because Widgets are publicly available we re-use the public ORB specified by the WidgetFactory_i servant. WidgetFactoryData stores all the information that’s shared across our “component”.

```
  int main(int argc,char* argv[]) {
    try {
      WidgetFactoryData data;
      data.factory_.orb_.initialize(
                             argc,argv,"public");
      data.factory_admin_.orb_.initialize(
                             argc,argv,"local");
```

Instantiate the “container” and initialize/activate the two independent ORBs

```
      data.factory_.attachToContainer(&data);
      data.factory_admin_.attachToContainer(&data);
      data.factory_.createServant();
      data.factory_admin_.createServant();
```

Attach the two factory servant types to the container and create the Objects, which will by virtue of the policies given automatically register themselves with the IOR table and the Naming Service.

```
      data.widget_poa_
        = POAPolicy::create_from_poa<
               POAPolicy::TransientSystemID<> >
              (data.factory_.orb_,
               data.factory_.poa_,
               "widget_poa");
```

Create the POA that Widget_Is will be activated in.

```
      data.event_.wait();
```

Wait for the WidgetFactoryAdmin_i servant to signal that the application should shut down. And then proceed to final cleanup.

```
      data.factory_admin_.destroy_all_widgets();
```

Destroy all Widget_i instances that were created on behalf of clients.

```
      data.widget_poa_->destroy(true,true);
```

Destroy the POA we manually created above

```
      data.factory_admin_.destroyServant();
      data.factory_.destroyServant();
```

Destroy the two factory servants

```
      data.factory_.orb_->destroy();
      data.factory_admin_.orb_->destroy ();
```

Finally, shutdown the two ORBs and their associated threads.

```
    }
    catch(const CORBA::Exception&) {return -1;}
    catch(...) {return -2;}
    return 0;
  }
```

5 Conclusion

With the aid of a policy-driven CORBA template library, the deployment of the machinery needed to build large-scale distributed object computing facilities has been greatly simplified. Expertise about the CORBA setup mechanisms has been encapsulated in an extensible and easily applied framework of policies, thus allowing rapid development that concentrates on the application’s functionality and lowers the barrier to entry for developers. Code reuse is promoted without relying on methods that are, at best, hard to extend (procedural, simple object-oriented inheritance or cut and paste). Therefore, the developer’s task is reduced to that of providing the application’s core functionality, freed of the burdens imposed by classic CORBA application designs. Easy extensibility ensures that future needs such as Real-Time ORB usage, SSLIOP setup, and Trading Service registration can be added once as policies and deployed with minimal effort.

Jeff Mirwaisi
jeff_mirwaisi@yahoo.com

6 References

[1] Michi Henning, Steve Vinoski, *Advanced CORBA Programming with C++*, 1999, Addison-Wesley
[2] Andrei Alexandrescu, *Modern C++ Design*, 2001, Addison Wesley
[3] Mark Delaney, “Parameterized Inheritance Considered Helpful” *CUJ*, January 2003
[4] David Vandevoorde, Nicolai M. Josuttis, *C++ Templates: The Complete Guide*, 2002, Addison Wesley
[5] Jim Coplien, “The Curiously Recurring Template Pattern”, *C++ Report*, Feb. 1995, 24-27