overload 82

DECEMBER 2007 £3

The Model Student
Journey into the world of the travelling salesman

The PfA Papers
We continue our descent into Parameterise From Above

Java Protocol Handlers
Using URLs to transparently access data from different places

Creating awareness Exposing problems
To develop better software we need to learn and organisations need to change

A magazine of ACCU

ISSN 1354-3172

OVERLOAD 82

December 2007

ISSN 1354-3172

Editor

Alan Griffiths
overload@accu.org

Advisors

Phil Bass
phil@stoneymanor.demon.co.uk

Richard Blundell
richard.blundell@gmail.com

Alistair McDonald
alistair@inrevo.com

Anthony Williams
anthony.ajw@gmail.com

Simon Sebright
simon.sebright@ubs.com

Paul Thomas
pthomas@spongelava.com

Ric Parkin
ric.parkin@ntlworld.com

Roger Orr
rogero@howzatt.demon.co.uk

Simon Farnsworth
simon@farnz.co.uk

Advertising enquiries

ads@accu.org

Cover art and design

Pete Goodliffe
pete@cthree.org

Copy deadlines

All articles intended for publication in Overload 83 should be submitted to the editor by 1st January 2008 and for Overload 84 by 1st March 2008.

ACCU

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

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

Overload is a publication of ACCU
For details of ACCU, our publications and activities, visit the ACCU website: www.accu.org

The Essence of Success

What makes a successful project?

Looking both ways

Last month you had a guest editor – Ric Parkin – one of the regular team that stepped into the editorial role whilst I went off on holiday. I'm pleased that things went smoothly, and I enjoyed the rest. In fact I enjoyed the rest so much that I've taken the opportunity for another one for the next issue: this time another team member – Roger Orr – is going to take over for an issue. I'm very confident that this too will go smoothly.

This seems very different from the situation this time last year when Ric, Roger and others first volunteered to help out with issue 76 and much closer to the 'golden age' story I told in that editorial:

> At the moment it seems like it was a long time ago and in a galaxy far, far away that lots of material was submitted to Overload by enthusiastic authors, an enthusiastic team of volunteer advisors reviewed it all, the editor swiftly passed their comments back to the authors who produced a new, much improved draft of the article which then was seen to be good by all and put in the magazine for your reading pleasure.

I'd like to thank everyone who contributed to realising this vision: the overload review team, the authors and my very understanding fiancée. Please keep up the good work!

What criteria identify success?

We often hear about the high rate of failure of software development projects. But are the right criteria being used to assess success? The usual measure quoted is 'on time, on budget, to specification' – and this ignores several important issues. Firstly, the specification is rarely nailed down adequately at the start of the project (and – far too often – not even at the end); secondly, the budget is rarely immovable either; and thirdly, except in rare cases (Y2K, statuary requirements, etc.) the timing isn't a business requirement.

There are good reasons why specifications are not generally fixed – most software projects are explorations of a both the problem and solution domains. While the high level requirements are often discoverable up front, the way in which they will met is only determined as the project progresses. The investigative nature of most projects also affects the budget and time scale.

Even though delivery dates, costs and functionality are written down at the start of projects (in many cases into contracts) it is very rare that they don't change. In the case of small changes no-one worries, in the case of large changes even contracts get renegotiated. But, is a project that doesn't meet its starting scope, budget and delivery date really a failure if it the customer gets what they need, when they need it and for a cost they can afford?

Well, there are other measures of success. One can look at whether the resources invested in a project could have been better deployed elsewhere: is it more profitable to spend time and money on better advertising or on better software? The value of the results of this work is the 'return on investment' and is a key factor in deciding whether a project is worthwhile. Generally, a business will only undertake projects that are expected to exceed a 'minimum return on investment' – and, if a project exceeds this, then it can be counted a partial success. (Of course, if it exceeds the expected return on investment it can be counted a total success.)

We all like to contribute to the profitability of the company that we work for. But for most of us that isn't our sole goal in life. There is a perspective that says a successful project is one that the participants would like to repeat. It is quite easy to imagine projects that are highly profitable, but where the developers on the project turn around and say 'I never want to do anything like that again'. This happens when their personal costs (in time, goodwill and opportunity) are greater than the rewards (in money and recognition).

These are not the only definitions of success – I once worked on a project that no-one enjoyed, went on for far longer than planned (and therefore cost far more than expected and, in particular, more than the customer paid), failed to deliver a key customer requirement (which happened to be impossible) and was still a 'success'. I did see the spreadsheet produce by the project manager concerned to justify this classification. In my more cynical moments I feel it owed more to the bonus structure he was working to than to anything else (several significant costs were not shown as part of the project budget), but it appears that his definition of success was accepted.

Most projects fail on some measures of success and succeed on others – I've just participated in a retrospective of a project that went far over the original time and budgets (but these were changed to reflect scope changes during the course of the project – it came in to their current values). It very narrowly passed a few regulatory milestone deliveries. The developers were all stressed by the lack of time to do things well – and there is a big build up of technical debt to be addressed. But there is a lot of value delivered to the business, and there is an appreciation of their achievement. I'd call it a successful project, but this would not qualify using the usual 'on time, on budget, to specification' criteria (or, for that matter, everyone wanting to 'do it again').

Alan Griffiths is an independent software developer who has been using "Agile Methods" since before they were called "Agile", has been using C++ since before there was a standard, has been using Java since before it went server-side and is still interested in learning new stuff. His homepage is http://www.octopull.demon.co.uk and he can be contacted at overload@accu.org

The right sort of success

Indeed it is easy to demonstrate that not all software development projects can be measured by the same criteria – one only has to consider a typical free/open source project like the Mozilla web browser. This project cannot be judged by the traditional 'on time, on budget, to specification' criteria. Admittedly the participants in this project could well be delivering benefits to their businesses (by implementing features or fixing bugs that allow them to provide value) and such contributions could be judged on this basis. However, closer examination shows this to be in error, it would be considerably easier and deliver the same value to a business to create an enhanced or fixed version that can be used internally – so why undertake the additional cost of getting a submission approved? And how could one fit the contributions of Toronto's Seneca College into this model?

One reason for highlighting the different flavours of success is that everyone should know which one matters in the current circumstances. When different participants in a project are seeking different goals none of these goals are likely to be achieved.

One of the identifying features of Agile Methods is the focus on identifying the piece of work that will deliver the best value (to the business) for development effort, and to tackle that next. The nature of the 'value' is deliberately left unspecified and assessed by the business concerned. Always allocating the work offering the best return is an easy way to communicate what is valuable: developers on these (and traditional) projects will naturally focus on the piece of work they are undertaking at the time and with this approach to planning are focussed on the work that gives the best return. In contrast traditional planning methods treat the project as a whole as delivering 'value' and only coarse classification of features into 'essential' or 'desirable' are even undertaken – with the possibility that high value items can end up at risk because of schedule overruns or resource overruns. (And when the project manager realised this, incomplete pieces of work will be abandoned in order to progress them – with all the pain and cost of context switching that this implies.)

However a project is planned, it is a good idea to know which (if any) features are fixed and which may be adjusted in scope without risking the success of the project itself.

So what about a project like Overload? How do we measure success? Clearly, the return is self-improvement: improving our knowledge of software development (or, in the case of authors, our ability to communicate what we know about it). Currently it feels like a success. What do you think?

Better late than never

At the last ACCU conference I ran a workshop on 'Agile Tools for C++' – the idea being to use the expertise available at the conference to collate knowledge about the wide range of alternative tools available. Why for C++? Well, as Russel Winder recently posted on accu-general:

> Unlike most languages where there are one or two standard [unit-testing] frameworks and everyone just uses them, C++ seems to generate a plethora so no-one has any idea which one to use (unless you are a member of one of the tribes of course).

What Russel claims about unit testing frameworks applies to a lot of other things too: comms libraries, editors, build systems, etc. Because none of us has the time and energy to invest in trying the whole range of tools I arranged the workshop and promised to write up the results. To my embarrassment, since the conference the notes produced by this workshop have been sitting in a file awaiting this happening.

Three things brought this to mind recently – firstly the next conference is getting closer and with it my sense of guilt has been increasing; secondly, Allan Kelly has just written up his session from the same conference; and thirdly, at XPDay I encountered one of the participants of the workshop who asked me what had happened to the write up.

Anyway – I haven't forgotten, I just haven't got around to it yet. Hopefully, in time for the next issue. (And yes, not having to edit the next issue has a lot to do with getting the article finished at last.)

Seasons comments

This is the Christmas issue of Overload, but unfortunately there is no seasonal article this year. But it is a good occasion for you to take a moment to reflect upon the last year and what has happened. The production of Overload has changed a lot for the better. I hope that the reading of Overload is providing benefits to you readers.

Merry Christmas to you all!

The PfA Papers: Context Matters

Continuing his history of Parameterise from Above, Kevlin Henny looks at Singletons and Context Objects.

The odyssey of the PARAMETERIZE FROM ABOVE (PFA) pattern started in 2001. It emerged from a recurring recommendation made during design consultancy [Henney2007a] and matured through its incorporation in the Programmer's Dozen collection of thirteen recommendations [Henney2007b]. The essence of the pattern is that parameterization is better done through explicit parameters at the point of configuration or call than through the automagic and coupling of global state.

In this third instalment of 'The PfA Papers', I will continue to chart the emergence and refinement of PARAMETERIZE FROM ABOVE by pursuing another strand in its history and development: the context object. (Although it may seem a little strange to chart the development of a pattern that has yet to be documented, things can exist in a culture without actually, err, needing to exist, so to speak.)

Context objects

The brute-force approach to capturing and passing execution context between components in a program sidesteps the question of passing altogether: represent the execution context as global state in the program.

If you are feeling a little guilty about having global variables in your code (sometimes they are called static variables just to assuage your guilt), you can always use SINGLETON – it may not do much for the quality of your code, but you get to accumulate another Gang-of-Four pattern-usage point! (Yes, there are teams and developers who do something like this and take it seriously.) However, whichever way you look at it, SINGLETON simply wraps up the global state and puts a nice face on it. SINGLETON doesn't address the deeper issues of coupling and hardwiring that are the real problem with global variables. And, for good measure, SINGLETON makes the basic problem harder by introducing new challenges, such as the subtleties involved in correctly initialising a SINGLETON in a multithreaded environment.

However, this is not just a random excuse to pick on the faults of SINGLETON and global variables. In a system where the code, the program state, the execution paths, etc., are all partitioned across projects, packages, classes and functions, objects, processors and threads, and so on, the notion of global state is less meaningful and less manageable. This is the root of the problem.

OK, so perhaps a little PFA would do the trick? Well, the smallest application of PFA you can have is to pass a single argument of simple type. So, it stands to reason that a slightly larger application of PFA is to pass two arguments, each of simple type. A little induction suggests that you can pass all the execution context you ever need by having a sufficiently long argument list. However, a little reality check tells you that this is not going to work out too well: long argument lists are a pain [Henney2006]. As Alan Perlis once put it [Perlis1982]:

> If you have a procedure with 10 parameters, you probably missed some.

The issue is that not only is a long argument list cumbersome, its content is inherently unstable. It is the wrong application of PFA: rather than PFA in sprinkles, you need to PFA in chunks. The solution is to recognise that the unit of stability is the concept of what the argument list represents – the context – and not the individual elements. What you parameterize should ideally be aligned with the concept as a whole and the unit of stability, which, in this case, means that you pass through an object representing the context, not a fragmented set of individual properties. Of course, that's not the end of the story: if you treat your context object as little more than a bucket of arbitrary parameters, you shouldn't be surprised when it degenerates into a bucket of arbitrary parameters.

From prehistory to the present

There are many ways to look at patterns: one is that a pattern is a recurring practice; another identifies the pattern as the documented description of said practice. The former view means that many patterns are in use without documentation or even agreement on a common name: they are just 'the way we do things here' or 'how I've always seen that problem solved most effectively'. The latter view emphasises that pattern authors do not own the patterns they describe: they own their descriptions. You cannot steal a pattern, only be inspired by its description or application.

Taking these two views together, it seems obvious that there can be many competing descriptions for the same practice. Some of these may emerge independently; some of these may be related by refinement or specialisation to a context. So it is with the CONTEXT OBJECT pattern. The story of CONTEXT OBJECT's description in pattern form is a little like a long wait for a bus, only to have two (or more) turn up at once.

In 2002 Allan Kelly started documenting what came to be called ENCAPSULATED CONTEXT and has since been included in Pattern Languages of Program Design 5 in 2006 [Manolescu+2006]. It was some of the discussion and feedback around the pattern that prompted me to write up CONTEXT ENCAPSULATION, a pattern language of four patterns that covered the design space of context objects – as it turned out, a small pattern language but a big paper. This paper was workshopped at the EuroPLoP conference in 2005 [Henney2005]. The root pattern description went on to form the basis of the CONTEXT OBJECT write-up in POSA4 [Buschmann+2007a].

This thread of history is interwoven with other threads, making the timeline a little more tangled. In September 2003 I was in Oslo attending the JavaZone conference. John Crupi, one of the authors of Core J2EE Patterns [Alur+2003], was presenting some of what was new in the second edition of that book, including – you guessed it – the CONTEXT OBJECT pattern. I was there presenting the Programmer's Dozen [Henney2007b],

Kevlin Henney is a long-standing member of ACCU, joining before it actually was ACCU and contributing to Overload when it was numbered in single digits. He recently co-authored two volumes in the Pattern-Oriented Software Architecture series, *A Pattern Language for Distributed Computing* and *On Patterns and Pattern Languages*. Kevlin can be contacted at kevlin@curbralan.com.

The idea of capturing execution context as an object or some kind of structure is perhaps older than you think

so the connection to PFA was reinforced, as was the commonality with Allan's work.

It turns out that 2005 was a busy year for CONTEXT OBJECT on the pattern conference front: I submitted CONTEXT ENCAPSULATION at EuroPLoP in Bavaria; Arvind Krishna, Doug Schmidt and Michael Stal wrote a version of CONTEXT OBJECT as a standalone pattern for PLoP in Illinois [Krishna+2005]; Uwe Zdun included CONTEXT OBJECT as one of a collection of patterns at VikingPLoP in Finland [Zdun2005]. Each paper took a different point of view and placed the pattern in a different context, but the recurrence and soundness of the practice was clearly established. And, because pattern papers are considered to be works in progress, later versions of each paper cross-referenced one another!

The idea of capturing execution context as an object or some kind of structure is perhaps older than you think. For example, in the 1970s the technique was used in Scheme's eval procedure to provide context for evaluation of an expression [Abelson+1984, Scheme]. It has also appeared in pattern form many times, but hidden as part of a larger pattern. For example, CONTEXT OBJECT is a key player in the INTERPRETER pattern [Gamma+1995], which, with hindsight, can be seen as a pattern compound comprising COMMAND, CONTEXT OBJECT and COMPOSITE Buschmann+2007b]. Likewise, CONTEXT OBJECT plays an essential role in the Interceptor pattern [Schmidt+2000]. More explicitly, PARAMETER OBJECTSs or ARGUMENTS OBJECTs have also previously been identified and described in pattern form [Noble1997, Fowler1999].

So, it's not just that two (or more) buses turn up at the same time: quite a few buses have already been past.

Encapsulated Context

Documenting a pattern is a journey of understanding, with snapshots taken along the way. One example of how things can change is the name. Allan's pattern description started life with the longer, imperative name of ENCAPSULATE EXECUTION CONTEXT. This was later shortened to ENCAPSULATE CONTEXT. The directive-based name then shifted to a noun phrase that described the resulting outcome, ENCAPSULATED CONTEXT.

My interest and involvement goes back to the ENCAPSULATE EXECUTION CONTEXT days, but what happened later was just as interesting [Henney2005]:

> It all started a couple of years ago – the summer of 2002, to be precise. Following a discussion thread on the ACCU's main list, Allan Kelly decided to document a pattern for addressing the problem of propagating context through a program that neither relied on the coupling of global variables and their kin nor on the unmanageability of long argument lists. I volunteered to shepherd the paper informally, which Allan then submitted it to EuroPLoP 2003. It received further shepherding from Frank Buschmann and was accepted and workshopped at the conference.
>
> There was recognition that the scope of the pattern was perhaps greater than could be contained conveniently within a single pattern. Most of the subtlety was in realizing the pattern effectively, and the options available and decisions that needed to be taken formed a long tail in the presentation of the pattern. Following its inclusion in the EuroPLoP proceedings [Kelly2003], the paper was also published in the ACCU's Overload magazine [Kelly2004], where it generated some heated discussion on the editorial review team [Overload2004] and the letters page [Overload2005]. Although ENCAPSULATED CONTEXT's description contained careful discussion of how to avoid having a context object turn into an uncohesive blob of code, this discussion sometimes appeared to be overlooked.

Allan managed to sum up the tension felt by someone struggling not just with the root problem that leads to ENCAPSULATED CONTEXT, but also in its effective application [Henney2005]:

> The devil is in the detail.
>
> You have this system...
>
> - Try globals... well, probably you don't: the one thing you learned in school was no globals.
> - You try for SINGLETON, it is in the book, it is good... but then you find you have these nasty ripples... then someone tells you it's a bad thing and it's obvious to you.
> - So you try passing parameters: they overwhelm you.
> - You refactor a bit (à la Fowler) and before you know it you've got ENCAPSULATED CONTEXT.
> - You carry on down this path, you get more mileage here, but over time it starts to look like Foote's BIG BALL OF MUD.
>
> The solution is to reduce the coupling, improve the cohesion, but how?

The question of how to deal with the BIG BALL OF MUD is an important consideration in applying the pattern. Don't mention it, and that will be considered an oversight or weakness of the pattern. However, documenting everything as equally significant is likely to overwhelm the reader. The core advice remains sound, but what is needed is separation.

Context Encapsulation

It was this question of communicating the core idea while still communicating the follow-on considerations, as well as the flurry of correspondence in Overload, that got me interested in looking at ENCAPSULATED CONTEXT from a different point of view [Henney2005]:

> Recasting the pattern in terms of a pattern language rather than a single pattern allows the flow of design issues to be identified more explicitly, promoting each of the considerations and decisions as a response to the issues raised in another pattern. Instead of considering the coupling and cohesion issues as simply being implementation details within a single pattern, the design process is made more explicit by naming and connecting some of these design decisions. Pulling support patterns out of a larger root pattern helps to manage pattern scope, which can otherwise creep with each new consideration that is incorporated.

At the beginning of 2005 I had been playing around with the idea of reasoning about pattern languages in terms of processes, which gave rise to a grammatical perspective, and relating this to pattern compounds and

pattern sequences. That sounds like a lot of abstract pattern theory, but does it have any practical application? The discussion about ENCAPSULATED CONTEXT came along at just this point. I realised that it might offer a particular vehicle for exploring the idea in a practical way: small enough to keep down to a handful of patterns and combinations; rich enough and real enough to illustrate the idea in practice.

The *Context Encapsulation* pattern language contains four patterns. The root of the language is ENCAPSULATED CONTEXT OBJECT, a name that blends ENCAPSULATED CONTEXT with CONTEXT OBJECT. The other three patterns in the language address the design decisions that can flow from an ENCAPSULATED CONTEXT OBJECT. These four protagonist patterns are summarised as follows [Henney2005]:

> ENCAPSULATED CONTEXT OBJECT: Pass execution context for a component, whether it is a layer or an individual object, as an object rather than as a long argument list of individual configuration parameters or implicitly as a global service. The execution context may include external configuration information and services such as logging.
>
> DECOUPLED CONTEXT INTERFACE: Reduce the coupling of a component to the concrete type of the ENCAPSULATED CONTEXT OBJECT by defining its dependency in terms of an interface, whether interface or INTERFACE CLASS, rather than the underlying implementation type. This allows substitution of alternative implementations, including NULL OBJECTS and MOCK OBJECT.
>
> ROLE-PARTITIONED CONTEXT: Split uncohesive ENCAPSULATED CONTEXT OBJECT interfaces into smaller more cohesive context interfaces based on usage role, each expressed with a DECOUPLED CONTEXT INTERFACE or through a ROLE-SPECIFIC CONTEXT OBJECT.
>
> ROLE-SPECIFIC CONTEXT OBJECT: Multiple context interfaces may be realized either together in a single object or with one object per role. The latter option allows independent parts of a context to be more loosely coupled and separately parameterized.

The devil may well be in the details, but the details are in the paper. 'The PfA Papers' is intended to shed light on the history and insights around PARAMETERIZE FROM ABOVE, so we should probably get back to the main act.

And so to PARAMETERIZE FROM ABOVE...

Given the undocumented state of the PARAMETERIZE FROM ABOVE pattern, it is perhaps noteworthy that tucked away in the notes at the end of CONTEXT ENCAPSULATION is a discussion of PfA and the relationship both to the patterns in the paper and elsewhere. The notes include the following summary of PARAMETERIZE FROM ABOVE:

> Within a layered system, some commonly used complex objects or simple values, used by different layers or by many different parts of a given layer, may find themselves expressed in global form, e.g. as SINGLETON or MONOSTATE objects. This parameterizes components from below, but hardwires their dependencies, increasing the coupling of the component and making alternatives difficult or impossible to substitute. Instead, invert the relationship, so that these objects are passed in from above, i.e. so that the calling or owning component in the layer above passes in the appropriate instance.

The relationship to CONTEXT OBJECT s is also made clear:

> Whether in terms of individual parameterizing integers or larger-scale, architecturally significant objects, to PARAMETERIZE FROM ABOVE represents a common reaction to unnecessary system-wide hardwiring of certain assumptions and facilities, and hence it is often employed in reaction to global variables, SINGLETONS, etc. Thus, an ENCAPSULATED CONTEXT OBJECT is an application of PARAMETERIZE FROM ABOVE, made more specific with respect to its context (sic) of application and its decomposition in terms of other patterns. ■

References

[Abelson+1984] Harold Abelson, Gerald Jay Sussman and Julie Sussman, *The Structure and Interpretation of Computer Programs*, MIT Press, 1984

[Alur+2003] Deepak Alur, John Crupi and Dan Malks, *Core J2EE Patterns*, 2nd edition, Addison-Wesley, 2003

[Buschmann+2007a] Frank Buschmann, Kevlin Henney and Douglas C Schmidt, *Pattern-Oriented Software Architecture*, *Volume 4: A Pattern Language for Distributed Computing*, Wiley, 2007

[Buschmann+2007b] Frank Buschmann, Kevlin Henney and Douglas C Schmidt, *Pattern-Oriented Software Architecture, Volume 5: On Patterns and Pattern Languages*, Wiley, 2007

[Fowler1999] Martin Fowler, *Refactoring*, Addison-Wesley, 1999

[Gamma+1995] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, *Design Patterns*, Addison-Wesley, 1995

[Henney2005] Kevlin Henney, 'Context Encapsulation', EuroPLoP 2005, July 2005, http://www.two-sdg.demon.co.uk/curbralan/papers/europlop/ContextEncapsulation.pdf

[Henney2006] Kevlin Henney, 'Long Argument Lists', *Reg Developer*, June 2006, http://www.regdeveloper.co.uk/2006/06/28/argument_lists/

[Henney2007a] Kevlin Henney, 'The PfA Papers: From the Top', *Overload 80*, August 2007, http://accu.org/index.php/journals/1411

[Henney2007b] Kevlin Henney, 'The PfA Papers: The Clean Dozen', *Overload 81*, October 2007, http://accu.org/index.php/journals/1420

[Kelly2003] Allan Kelly, 'Encapsulate Context', EuroPLoP 2003, June 2003

[Kelly2004] Allan Kelly, 'Encapsulate Context', *Overload 63*, October 2004, http://accu.org/index.php/journals/246

[Krishna+2005] Arvind S Krishna, Douglas C Schmidt and Michael Stal, 'Context Object', September 2005, http://www.dre.vanderbilt.edu/~arvindk/Context-Object-Pattern.pdf

[Manolescu+2006] Dragos Manolescu, Markus Voelter and James Noble (editors), *Pattern Languages of Program Design 5*, Addison-Wesley, 2006

[Noble1997] James Noble, 'Arguments and Results', *The Computer Journal*, 1997, http://citeseer.ist.psu.edu/107777.html

[Overload2004] 'Editorial', *Overload 64*, December 2004, http://accu.org/index.php/journals/249

[Overload2005] 'Letters to the Editor', *Overload 65*, February 2005, http://accu.org/index.php/journals/259

[Perlis1982] Alan J Perlis, 'Epigrams in Programming', *ACM SIGPLAN*, September 1982, http://www.cs.yale.edu/quotes.html

[Scheme] 'Scheme in Scheme', http://academic.evergreen.edu/curricular/fofc00/eval.html

[Schmidt+2000] Douglas C Schmidt, Michael Stal, Hans Rohnert and Frank Buschmann, *Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects*, Wiley, 2000

[Zdun2005] Uwe Zdun, 'Patterns of Argument Passing', VikingPLoP 2005, September 2005, http://wi.wu-wien.ac.at/~uzdun/publications/arguments.pdf

The Model Student

Richard Harris begins a series of articles exploring some of the mathematics of interest to those modelling problems with computers. Part 1: The Regular Travelling Salesman.

The travelling salesman problem, or TSP, must be one of the most popular problems amongst computer science students. It is extremely simple to state; what is the shortest route by which one can tour n cities and return to one's starting point? Figure 1 shows random and optimal tours of a 9-city TSP.

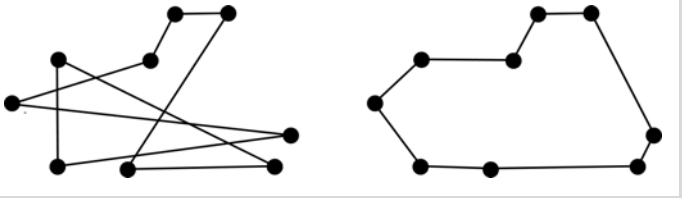

Figure 1

On first inspection, it seems to be fairly simple. The 9-city tour in figure 1 can be solved by eye in just a few seconds.

However, the general case is fiendishly difficult. So much so that finding a fast algorithm to generate the optimal tour, or even proving that no such algorithm exists, will net you $1,000,000 from the Clay Mathematics Institute [Clay].

This is because it is an example of an NP-complete (nondeterministic polynomial complexity) problem. This is the class of problems for which the answer can be checked in polynomial time, but for which finding it has unknown complexity. The question of whether an NP-complete problem can be solved in polynomial time is succinctly expressed as 'is P equal to NP?' and answering it is one of the Millennium Prize Problems, hence the substantial cash reward.

The answer to this question is so valuable because it has been proven that if you can solve one NP-complete problem in polynomial time, you can solve them all in polynomial time. And NP-complete problems turn up all over the place.

For example, secure communication on the internet relies upon $P \neq NP$. The cryptographic algorithms used to make communication secure depend on functions that are easy to compute, but hard to invert. If P=NP then no such functions exist and secure communication on an insecure medium is impossible. Since every financial institution relies upon such communication for transferring funds, I suspect that you could raise far more than $1,000,000 if you were able to prove that P=NP. Fortunately for the integrity of our bank accounts the evidence seems to indicate that if the question is ever answered it will be in the negative.

So, given that some of the keenest minds on the planet have failed to solve this problem, what possible insights could an amateur modeller provide?

Not many, I'm afraid. Well, not for this problem exactly.

I'd like to introduce a variant of the TSP that I'll call the regular travelling salesman problem. This is a TSP in which the cities are located at the vertices of a regular polygon. Figure 2 shows the first four regular TSPs.

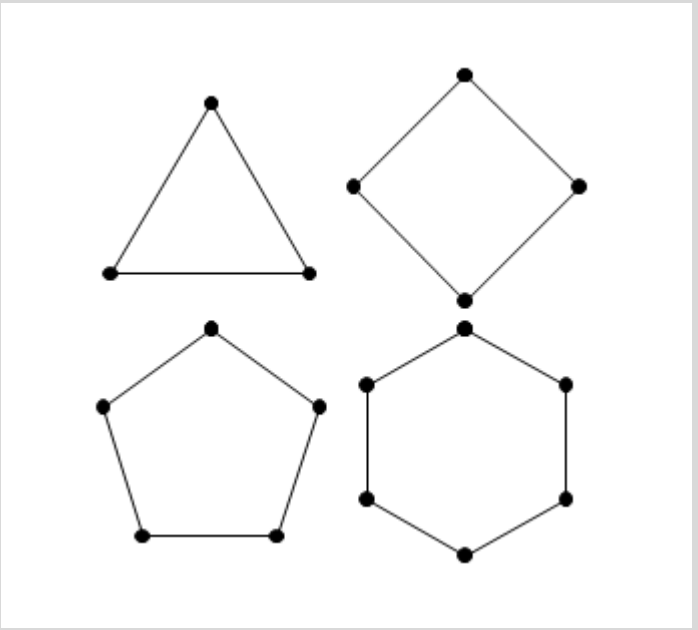

Figure 2

The question of which is the shortest tour is rather uninteresting for the regular TSP as it's simply the circumference of the polygon. Assuming that the cities are located at unit distance from the centre of the polygons (i.e. the polygons have unit radius), the length of the optimal tour can be found with a little trigonometry. Figure 3 shows the length of a side.

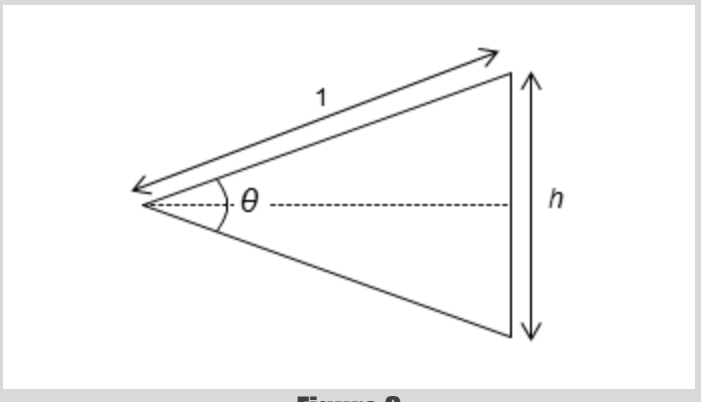

Figure 3

Richard Harris Richard has been a professinal programmer since 1996. He has a background in Artificial Intelligence and numerical computing and is currently employed writing software for financial regulation.

So are there any remotely interesting questions we can ask about the regular TSP?

For a tour of n cities, the length, l, of the optimal tour is given by:

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

$$\sin\frac{\theta}{2} = \frac{h}{2}$$

$$l = n \times h = 2n\sin\frac{\pi}{n}$$

As n gets large, so θ gets small, and for small θ, $\sin\theta$ is well approximated by θ itself. We can conclude, therefore, that for large n, the length of the optimal tour is approximately equal to 2θ. This shouldn't come as much of a surprise since for large n a polygon is a good approximation for a circle. In fact, it was this observation that Archimedes [Archimedes] used to prove that

> The ratio of the circumference of any circle to its diameter is less than $^{31}/_{7}$ but greater than $^{310}/_{71}$.

It's also fairly easy to find the length of the most sub-optimal tour. The key is to note that for odd n the furthest two cities from any given city are those connecting the opposite side of the polygon. Figure 4 shows the longest single steps and tour in a 5-city regular TSP.

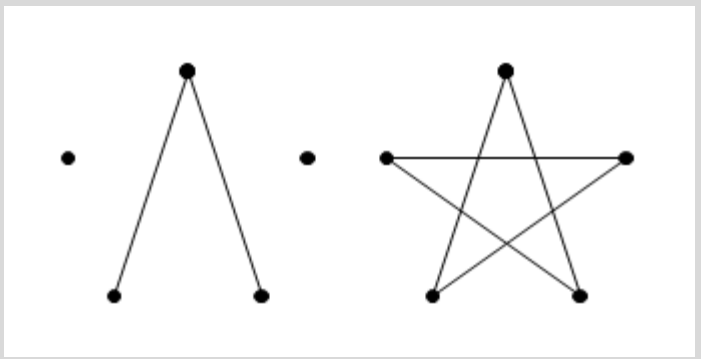

Figure 4

For odd regular TSPs, we can take a step of this length for every city, giving us a star shaped tour. We can calculate the length of this tour in a similar way to that we used to calculate the shortest tour length. Figure 5 shows the length of the longest single step.

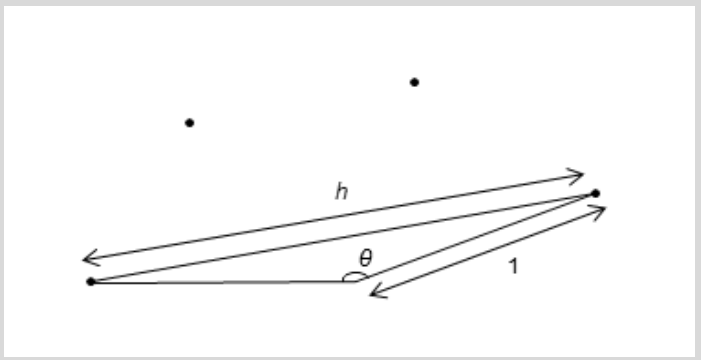

Figure 5

So for a tour of odd n cities, the length, l', of the worst tour is given by:

$$\theta = \frac{2\pi}{n} \times \frac{n-1}{2} = \pi - \frac{\pi}{n}$$

$$\sin\frac{\theta}{2} = \frac{h}{2}$$

$$l' = n \times h = 2n\sin\left(\frac{\pi}{2} - \frac{\pi}{2n}\right) = 2n\cos\frac{\pi}{2n}$$

This time for sufficiently large n, θ is small enough that $\cos\theta$ is well approximated by *1*. For large odd n, therefore, the length of the worst tour is approximately equal to $2n$. Once again, we could have equally well concluded this from the fact that for large n the polygon is a good approximation for a circle for which the largest step is across the diameter.

For an even number of cities the worst single step is to the city on the opposite side of the polygon with a distance of 2. Unfortunately each time we take such a step we rule it out for the city we visit, which will have to take a shorter step. So we can have $\frac{1}{2}n$ steps of length 2 and $\frac{1}{2}n$ steps of length strictly less than 2, giving a total length strictly less than $2n$.

This doesn't show that for an even number of cities the limit is $2n$, just that it cannot exceed $2n$. However, we can follow the longest step with the second longest to one of the first city's neighbours. We can repeat this for all but the last pair of cities for which we can take the longest step followed by the shortest. Figure 6 shows a $2n$-2 limit tour for 6 cities.

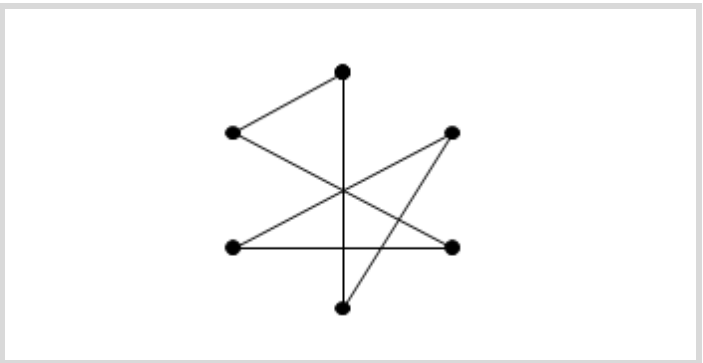

Figure 6

Whilst I haven't shown that this is the worst strategy, it does have a limit close to $2n$ for large n. It takes $\frac{1}{2}n$ steps with length 2, $\frac{1}{2}n$-1 steps with length approximately equal to 2 and one step with length approximately equal to 0 giving a total of $2n$-2.

So are there any remotely interesting questions we can ask about the regular TSP?

How about what the average length of a tour is? Or, more generally, how are the lengths of random regular TSP tours distributed?

This is where the maths gets a little bit tricky, so we'll need to write a program to enumerate the tours directly. The simplest way to do this is to

We can exploit the fact that our cities are located at the vertices of regular polygons

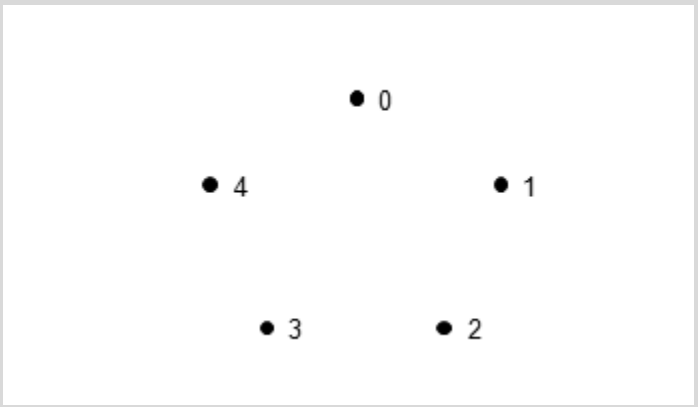

Figure 7

assign each city a number from 0 to *n*-1 so we can represent a tour as a sequence of integers. Figure 7 shows labels for a 5-city regular TSP.

A tour can be defined as:

```
#include <vector>
  namespace tsp
  {
    typedef std::vector<size_t> tour;
  }
```

We'll need some code to calculate the distance between the cities. We can save ourselves some work if we calculate the distances in advance rather than on the fly. We can exploit the fact that our cities are located at the vertices of regular polygons by noting that due to rotational symmetry the distance between two cities depends only on how may steps round the circumference separate them. Listing 1 shows a class to calculate distances between cities.

```
namespace tsp
{
  class distances
  {
  public:
    distances(size_t n);

    size_t size() const;
    double operator()(size_t step) const;
    double operator()(size_t i, size_t j) const;

  private:
    typedef std::vector<double> vector;

    size_t n_;
    vector distances_;
  };
}
```

Listing 1

```
tsp::distances::distances(size_t n) : n_(n),
   distances_(n)
{
  if(n_<3) throw std::invalid_argument("");

  static const double pi = acos(0.0) * 2.0;
  double theta = 2.0*pi / double(n_);
  double alpha = 0.0;

  vector::iterator first = distances_.begin();
  vector::iterator last = distances_.end();

  while(first!=last)
  {
    *first = 2.0 * sin(alpha/2.0);
    ++first;
    alpha += theta;
  }
}
```

Listing 2

The constructor does most of the work, calculating the distances between cities 0 to *n*-1 steps apart. Listing 2 calculates the distances between cities.

Of course we could have also exploited the reflectional symmetry that means the distance between cities separated by *i* and *n-i* steps are also the same, but I'm not keen to make the code more complex for the relatively small improvement that results.

The code to retrieve the distance between two cities is relatively simple and is shown in Listing 3.

```
double
tsp::distances::operator()(size_t step) const
{
  if(step>=distances_.size())
     throw std::invalid_argument("");
  return distances_[step];
}

double
tsp::distances::operator()(size_t i,
                           size_t j) const
{
  if(i>=n_ || j>=n_)
     throw std::invalid_argument("");
  return (*this)((i>j) ? i-j : j-i);
}
```

Listing 3

unless we specify otherwise, we'll use twice as many buckets as we have vertices

```
double
tsp::tour_length(const tour &t,
                 const distances &d)
  {
  if(t.size()==0)
     throw std::invalid_argument("");

  tour::const_iterator first = t.begin();
  tour::const_iterator next = first+1;
  tour::const_iterator last = t.end();

  double length = 0.0;
  while(next!=last) length += d(*first++,
                                *next++);
  length += d(*first, t.front());

  return length;
}
```

Listing 4

To calculate the length of a tour we need only iterate over it and sum the distances of each step (Listing 4, overleaf).

The final thing we'll need before we start generating tours is some code to keep track of the distribution of tour lengths. Listing 5 shows a class to maintain a histogram of tour lengths.

Most of the member functions of our histogram class are pretty trivial, so we'll just look at the interesting ones. Firstly, the constructors. Listing 6 shows how the tour histograms are constructed.

```
tsp::tour_histogram::tour_histogram(
   size_t vertices) :

vertices_(vertices),

histogram_(2*vertices)
  {
    init();
  }

tsp::tour_histogram::tour_histogram(
   size_type vertices,
   size_type buckets) :

vertices_(vertices),

histogram_(buckets)
{
  init();
}
```

Listing 6

```
namespace tsp
{
  class tour_histogram
  {
  public:
    struct value_type
    {
      double length;
      size_t count;
      value_type();
      value_type(double len, size_t cnt);
    };

    typedef std::vector<value_type>
       histogram_type;
    typedef histogram_type::
       size_type size_type;
    typedef const value_type &
       const_reference;
    typedef histogram_type::
       const_iterator const_iterator;

    tour_histogram();
    explicit tour_histogram(size_t vertices);
    tour_histogram(size_type vertices,
                   size_type buckets);

    bool      empty() const;
    size_type size() const;
    size_type vertices() const;

    const_iterator begin() const;
    const_iterator end() const;
    const_reference operator[](
       size_type i) const;
    const_reference at(size_type i) const;
    void add(double len, size_t count = 1);

  private:
    void init();
    size_type      vertices_;
    histogram_type histogram_;
  };
}
```

Listing 5

As you can see, unless we specify otherwise, we'll use twice as many buckets as we have vertices. This is because we've already proven that the maximum tour length is bounded above by $2n$, so it makes sense to restrict our histogram to values between 0 and $2n$ and dividing this into unit length ranges is a natural choice.

The problem is that the number of tours grows extremely rapidly with the number of cities.

```
void
tsp::tour_histogram::init()
{
  if(empty()) throw std::invalid_argument("");

  double step =
     double(2*vertices_) / double(size());
  double length = 0.0;

  histogram_type::iterator first
     = histogram_.begin();
  histogram_type::iterator last
     = histogram_.end();
  --last;

  while(first!=last)
  {
    length += step;
    *first++ = value_type(length, 0);
  }

  *first = value_type(double(2*vertices_), 0);
}
```

Listing 6

The `init` member function simply initialises the ranges for each of the buckets and sets their counts to 0. Note that, for our default histogram, the bucket identified with length *l* records tours of length greater than or equal to *l*-1 and less than *l*.

Listing 6 shows initialising the tour histogram.

We can exploit the fact that our histogram buckets are distributed evenly over the range 0 to 2*n* when recording tour lengths. To identify the correct bucket we need only take the integer part of the tour length multiplied by the number of buckets and divided by 2*n*.

Listing 7 shows adding a tour to the histogram

Now we have all of the scaffolding we need to start measuring the properties of random tours of the regular TSP. Before we start, however, we should be mindful of the enormity of the task we have set ourselves.

```
void
tsp::tour_histogram::add(double len)
{
  size_type offset((double(size())*len)/
     histogram_.back().length);
  if(offset>=size())
     throw std::invalid_argument("");
  histogram_[offset].count += 1;
}
```

Listing 7

The problem is that the number of tours grows extremely rapidly with the number of cities. For a TSP with *n* cities, we have a total of *n*! tours which are going to take a lot of time to enumerate.

Table 1 shows the growth of *n*! with *n*.

n	n!
3	6
4	24
5	120
6	720
7	5040
8	40320
9	362880
10	3628800

Table 1

We can improve matters slightly by considering symmetries again.

Firstly we have a rotational symmetry, in that we can start at any of the cities in a given tour and generate a new tour. By fixing the first city, we improve matters by a factor of *n*.

Secondly we have a reflectional symmetry in that we can follow any given tour backwards and get a new tour. By fixing which direction we take around the polygon, we reduce the complexity of the problem by a further factor of 2.

Whilst exploiting the rotational symmetry is relatively straightforward, the reflectional symmetry once again requires quite a bit of house-keeping. Hence I shall only attempt to exploit the former for the time being.

The first thing we're going to need is a way to generate the initial tour.

```
void
tsp::generate_tour(tour::iterator first,
                   tour::iterator last)
{
  size_t i = 0;
  while(first!=last) *first++ = i++;
}
```

Once we can do that it is a simple matter of iterating through each of the remaining tours and adding their lengths to our histogram. Fortunately there's a standard function we can use to iterate through them for us; `std::next_permutation`. This takes a pair of iterators and transforms the values to the lexicographically next largest permutation, returning false if there are no more permutations.

```
void
  tsp::full_tour(tour_histogram &histogram)
  {
    distances dists(histogram.vertices());
    tour t(histogram.vertices());
    generate_tour(t.begin(), t.end());

    do histogram.add(tour_length(t, dists));
    while(std::next_permutation(t.begin()+1,
                                t.end()));
  }
}
```

Listing 8

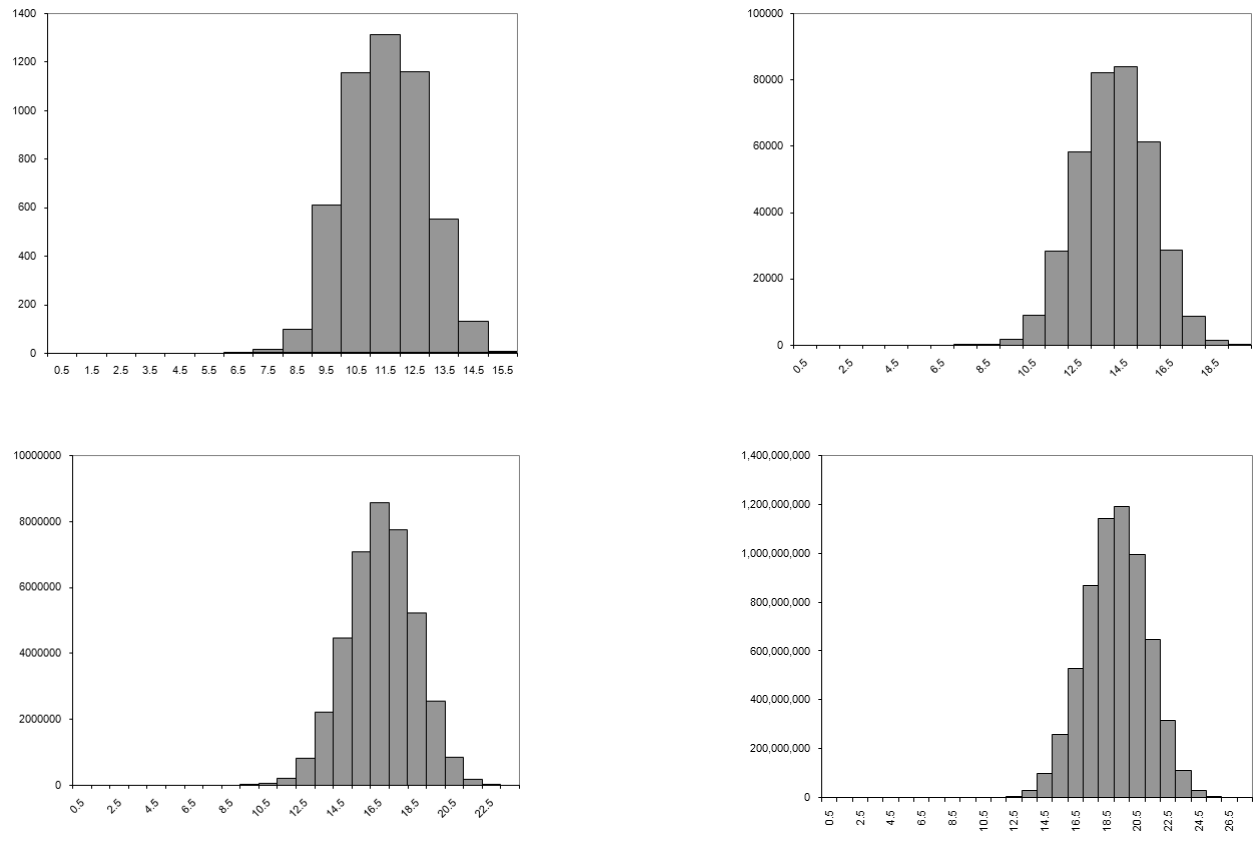

Figure 8

Using this function to calculate the histogram of tour lengths is relatively straightforward, as shown in Listing 8.

Note that exploiting the rotational symmetry of the starting city is achieved by simply leaving out the first city in our call to `std::next_permutation`.

Now we are ready to start looking at the results for some tours, albeit only those for which the computational burden is not too great.

Figure 8 shows the tour histograms for 8, 10, 12 and 14 city regular TSPs.

We can also use the histograms to calculate an approximate value for the average length of the tours. We do this by assuming that every tour that is added to a bucket has length equal to the mid-point of the range for that bucket. For our default number of buckets, this introduces an error of at most 0.5, which for large n shouldn't be significant. If you're not comfortable with this error it would not be a particularly difficult task to adjust the add member function to also record the sum of the tour lengths with which you could more accurately calculate the average length. I'm not going to bother though.

The approximate average lengths of the above tours, as both absolute length and in proportion to the number of cities, are given in Table 2.

The distributions shown by the histograms and the average tour lengths both hint at a common limit for large n, but unless we can analyse longer tours we have no way of confirming this. Unfortunately, the computational expense is getting a little burdensome as Table 3 illustrates.

n	?	? /n
8	10.99	1.37
10	13.51	1.35
12	16.07	1.34
14	18.62	1.33

Table 2

n	time (seconds)
8	0.002
10	0.180
12	22.140
14	4024.410

Table 3

So can we reduce the computational expense of generating the tour histograms? Well that, I'm afraid, is a question that shall have to wait until next time. ■

Acknowledgements

With thanks to Larisa Khodarinova for a lively discussion on group theory that lead to the correct count of distinct tours and to Astrid Osborn and John Paul Barjaktarevic for proof reading this article.

References & Further Reading

[Clay] Clay Mathematics Institute Millennium Problems, http://www.claymath.org/millennium.

[Archimedes] Archimedes, *On the Measurement of the Circle*, c. 250–212BC.

[Beardwood59] Beardwood, Halton and Hammersley, The Shortest Path Through Many Points, Proceedings of the Cambridge Philosophical Society, vol. 55, pp. 299-327, 1959.

[Jaillet93]Jaillet, Analysis of Probabalistic Combinatorial Optimization Problems in Euclidean Spaces, Mathematics of Operations Research, vol. 18, pp. 51-71, 1993.

[Agnihothri98] Agnihothri, A Mean Value Analysis of the Travelling Repairman Problem, IEE Transactions, vol. 20, pp. 223-229, 1998.

[Basel01] Basel and Willemain, Random Tours in the Travelling Salesman Problem: Analysis and Application, Computational Optimization and Applications, vol. 20, pp. 211-217, 2001.

[Hoffman96] Hoffman and Padberg, Travelling Salesman Problem, Encyclopedia of Operations Research and Management Science, Gass and Harris (Eds.), Kluwer Academic, Norwell, MA, 1996.

Functional Programming Using C++ Templates (Part 2)

Continuing his exploration of functional programming and template metaprogramming, Stuart Golodetz looks at some data structures.

Introduction

In my last article [Golodetz], I explored some of the similarities between functional programming in languages such as Haskell, and template metaprogramming in C++. This time, we'll look at how to implement compile-time binary search trees (see Figure 1 for an example). As mentioned previously, these are useful because they allow us to implement static tables that are sorted at compile-time.

Figure 1 is an example of a binary search tree. Note that an in-order walk of the tree produces a list which is sorted in ascending order.

Orderings

Our finished tree code will implement a compile-time map from a key type to a value type. Since binary search trees are sorted (an in-order walk of the tree will produce a sorted list), we'll need to define an ordering on the key type to do this.

We'll start by defining a default ordering of our various element types. Bearing in mind our definitions of the `Int` and `Rational` types in the last article, we can write this as follows:

```
template <typename x, typename y> struct Order
{
  enum { less = x::value < y::value };
  enum { greater = x::value > y::value };
};
```

We'll also define ordering predicates for later use with `Filter` (Listing 1).

```
template <typename y,
   template <typename,typename> class Ordering>
struct LessPred
{
  template <typename x>
  struct Eval
  {
    enum { value = Ordering<x,y>::less };
  };
};

template <typename y,
   template <typename,typename> class Ordering>
struct GreaterPred
{
  template <typename x>
  struct Eval
  {
    enum { value = Ordering<x,y>::greater };
  };
};
```

Listing 1

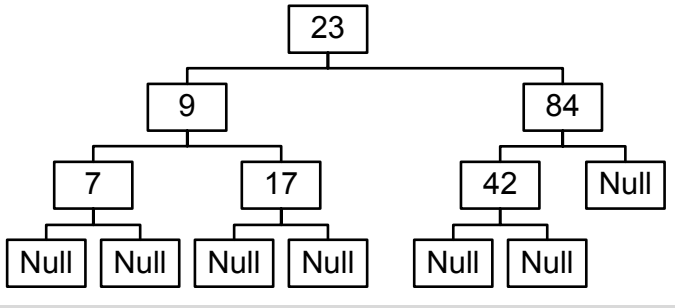

Figure 1

As we'll see later, our trees will be constructed from a list of (key,value) pairs. We can define a simple compile-time pair type as follows:

```
template <typename F, typename S>
struct Pair
{
  typedef F first;
  typedef S second;
};
```

The above definition is for a generic pair (not just one which can be used with our tree code). We can define an ordering for a (key, value) pair as follows (note that I've called this `KeyOrder` rather than defining it as a partial specialization of the `Order` template – we don't really want there to be a 'default' ordering for pairs):

```
template <typename x,
   typename y> struct KeyOrder;

template <typename k1, typename v1,
   typename k2, typename v2>
struct KeyOrder<Pair<k1,v1>, Pair<k2, v2> >
{
  enum { less = Order<k1,k2>::less };
  enum { greater = Order<k1,k2>::greater };
};
```

Building a balanced tree

For a binary search tree to be 'efficient', we want it to be as balanced as possible; in other words, we ideally want its height to be logarithmic rather

Stuart Golodotz Stuart has been programming for 13 years and is currently studying for a computing doctorate at Oxford University. His next project involves the geometric modelling of kidney cancer. He can be contacted at stuart.golodetz@comlab.ox.ac.uk

than linear in the number of its elements. A good way to go about arranging this when constructing a tree is to start from a list l of (key,value) pairs $(key_1, value_1)$ ... $(key_n, value_n)$, where all the key_i values are distinct, and proceed as follows:

1. Choose a pair $(key_s, value_s)$ from l such that:
$$\left|\#\{i : key_i < key_s\} - \#\{i : key_i > key_s\}\right|$$
is minimised, i.e. such that key_s divides the list of keys as evenly as possible. (Note that < and > in the above are defined in terms of the ordering on the key type.)
2. Generate two sublists:
$$l_L = \{(key_i, value_i) : key_i < key_s\}$$
and
$$l_R = \{(key_i, value_i) : key_i > key_s\}$$
(Note that $l_L \cup l_R \cup \{(key_s, value_s)\} = l$).
3. Recursively construct a tree for each of the sublists (unless they are the empty list) and attach the generated trees as the children of the current node. Set the *splitter* at the current node to be $(key_s, value_s)$.

```
template <typename xs,
   template <typename,typename> class Ordering>
struct BuildNode;

template <template <typename,typename>
   class Ordering>
struct BuildNode<NullType, Ordering>
{
  typedef NullType result;
};

template <typename k, typename v, typename xs,
   template <typename,typename> class Ordering>
struct BuildNode<List<Pair<k,v>,xs>, Ordering>
{
  typedef List<Pair<k,v>,xs> list;
  typedef typename ChooseSplitter<list,list,
     Ordering>::result splitter;

  typedef typename Filter<LessPred<splitter,
     Ordering>::Eval, list>::result leftList;
  typedef typename Filter<GreaterPred<splitter,
     Ordering>::Eval, list>::result rightList;

  typedef typename BuildNode<leftList,
     Ordering>::result leftChild;
  typedef typename BuildNode<rightList,
     Ordering>::result rightChild;

  typedef TreeNode<splitter, leftChild,
     rightChild> result;
};
```

Listing 2

Modulo minor alterations for different scenarios, this algorithm is a standard way of building (reasonably) balanced trees. (A similar sort of approach is used when building binary space partitioning (BSP) trees or decision trees.) How do we go about coding it using templates?

Conceptually, it's probably easiest to use a top-down approach to the problem in this instance (especially given the way the description above is written). We start by defining the **`BuildNode`** template. This takes a list of (key, value) pairs and an ordering, and defines a typedef called **`result`** which will be the type defining the resulting tree (Listing 2).

The way this works is exactly as you'd expect. First we choose a splitter, then we filter the list l (**`list`** in the code) twice to get l_L (**`leftList`**) and l_R (**`rightList`**). Finally, we recursively build the subtrees and combine them into a **`TreeNode`**.

When it comes to looking up values in the tree (which we will see later), we will need to know what ordering was used when building it. The **`BuildTree`** template uses **`BuildNode`** to build the actual tree, then wraps the tree and the ordering into a compound type:

```
template <typename xs, template <typename,
   typename> class Ordering = KeyOrder>
struct BuildTree
{
  typedef Tree<typename BuildNode<xs,Ordering>::
     result,Ordering> result;
};
```

Choosing a splitter

We now need to decide how to choose a splitter for a list. The basics of the idea were given in the high-level description above; the way we implement it in practice is to evaluate a metric for each element of the list and pick the one with the lowest score (or any one of those with the lowest score, if necessary). The metric computes the absolute difference between the number of elements less than the potential splitter and the number of those greater than it (Listing 3).

The **`ChooseSplitter`** template then uses this to pick the best possible splitter each time (Listing 4).

Outputting a tree

We have code for building a tree, but before we can test it we need some way of outputting trees. The code in Listing 5 does exactly that, indenting each layer of the tree by a fixed amount to make it easier to read.

An example

Figure 2 shows a key -> value map, implemented using a binary search tree. Null nodes are not shown.

Now that we can output trees, we can examine a concrete example of tree-building. Specifically, we will build the tree shown in Figure 2.

First of all, we define some macros to make the final code easier to read (see Listing 6).

```
template <int n>
struct Abs
{
  enum { value = n >= 0 ? n : -n };
};

template <typename y, typename xs,
   template <typename,typename> class Ordering>
struct EvaluateMetric;

template <typename y,
   template <typename,typename> class Ordering>
struct EvaluateMetric<y, NullType, Ordering>
{
  enum { less = 0, greater = 0 };
};

template <typename y, typename x, typename xs,
   template <typename,typename> class Ordering>
struct EvaluateMetric<y, List<x,xs>, Ordering>
{
  enum { less = Ordering<y,x>::less +
     EvaluateMetric<y,xs,Ordering>::less };
  enum { greater = Ordering<y,x>::greater +
     EvaluateMetric<y,xs,Ordering>::greater };
  enum { value = Abs<(greater - less)>::value };
};
```

Listing 3

```
template <typename ys, typename xs,
   template <typename,typename> class Ordering>
struct ChooseSplitter;

template <typename y, typename xs,
   typename Ordering>
struct ChooseSplitter<List<y,NullType>, xs,
   Ordering>
{
  typedef y result;
  enum { metric = EvaluateMetric<y,xs,Ordering>::
     value };
};

template <typename y, typename ys, typename xs,
   template <typename,typename> class Ordering>
struct ChooseSplitter<List<y,ys>, xs, Ordering>
{
  typedef typename ChooseSplitter<ys,xs,
     Ordering>:: result candidate;
  enum { candidateMetric = ChooseSplitter<ys,xs,
     Ordering>:: metric };
  enum { metric = EvaluateMetric<y,xs,Ordering>::
     value };
  typedef typename Select<(
     metric < candidateMetric), y, candidate>::
     result result;
};
```

Listing 4

```
template <typename T, int offset = 0>
struct OutputNode;

template <int offset>
struct OutputNode<NullType, offset>
{
  void operator()()
  {
    for(int i=0; i<offset; ++i) std::cout << ' ';
    std::cout << "Null\n";
  }
};

template <typename Splitter, typename Left,
   typename Right, int offset>
struct OutputNode<TreeNode<Splitter, Left, Right>,
   offset>
{
  void operator()()
  {
    for(int i=0; i<offset; ++i) std::cout << ' ';
    std::cout << Splitter::first::value << ' ' <<
       Splitter::second::value << '\n';
    OutputNode<Left, (offset+2)>()();
    OutputNode<Right, (offset+2)>()();
  }
};
template <typename Tree> struct OutputTree;

template <typename Root,
   template <typename,typename> class Ordering>
struct OutputTree<Tree<Root,Ordering> >
{
  void operator()()
  {
    OutputNode<Root>()();
  }
};
```

Listing 5

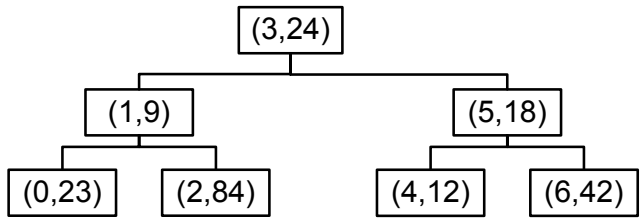

Figure 2

The KPLIST*n* macros take a key type and a value type, followed by a list of (key,value) pairs. Given the above definitions, we can build and output our tree as follows:

```
OUTPUT_TREE(BUILD_TREE(KPLIST7(Int,Int,0,23,1,9,
2,84,3,24,4,12,5,18,6,42)));
```

```
#define KPLIST1(kt,vt,k1,v1)              List<Pair<kt<k1>,vt<v1> >, NullType>
#define KPLIST2(kt,vt,k1,v1,k2,v2)        List<Pair<kt<k1>,vt<v1> >, KPLIST1(kt,vt,k2,v2)>
#define KPLIST3(kt,vt,k1,v1,k2,v2,k3,v3) List<Pair<kt<k1>,vt<v1> >, KPLIST2(kt,vt,k2,v2,k3,v3)>
...
#define KPLIST7 ...

#define BUILD_TREE(L)                     BuildTree<L>::result
#define OUTPUT_TREE(T)                    OutputTree<T>()()
```

Listing 6

This gives the expected result:

```
3 24
  1 9
    0 23
      Null
      Null
    2 84
      Null
      Null
  5 18
    4 12
      Null
      Null
    6 42
      Null
      Null
```

Lookup

Tree-building is all well and good, but to finish implementing compile-time maps, we now need to implement a lookup function on our trees. This turns out to be relatively straightforward: we locate the key we want to find in the tree recursively by comparing it against the key at the current node at each stage; if it's less than the nodal key, we recurse down to the left child, if it's greater than the nodal key, we recurse down to the right child, and if it's equal then we've found what we're looking for and 'return' the nodal value by means of a **`typedef`**. If the current node is null, the key can't be found in the tree and we 'return' null (or **`NullType`**). The code is shown in Listing 7.

One slight trick lies in the way we handle orderings: the ordering stored with the tree itself is an ordering over (key,value) pairs, since it was an ordering over the list from which the tree was built. The ordering we want for lookup, on the other hand, is an ordering over the key type. There are two equivalent ways of dealing with this: either we initially define an ordering on the key type and lift it to (key,value) pairs, or we initially define one on (key,value) pairs and lower it back to the key type. The first way probably makes more intuitive sense (since the alternative relies on the fact that the (key,value) pair ordering we define depends only on the key) and is left as a mini-exercise for the reader. In the code in Listing 7, we use the **`SubsidiaryOrdering`** template to lower the (key,value) pair ordering back to the key type; its definition is as shown in Listing 8.

Having thus finished our lookup implementation, we can try it out, again defining some macros to make things neater:

```
#define LOOKUP(kt,kv,T)Lookup<kt<kv>,T>::result
#define OUTPUT_VALUE(v)OutputValue<v>()()

...

OUTPUT_VALUE(LOOKUP(Int,4,BUILD_TREE(KPLIST7(
Int,Int,0,23,1,9,2,84,3,24,4,12,5,18,6,42))));
```

This correctly outputs the value 12, as we expect.

Conclusion

Let's take a step back from the code-face and take a look at what we've gained from this. Aside from seeing how to implement a compile-time map, which is a valuable technique in itself, we've seen that template metaprogramming in general can be a really useful technique, and (as discussed in the last article) one that needn't unnecessarily befuddle us if we keep in mind its similarities to functional programming in other languages. Aside from providing an interesting programming challenge in its own right, template metaprogramming has applications which are useful in the real world and is well worth looking into. ■

References

[Golodetz] Functional Programming Using C++ Templates (Part 1), Overload 81, October 2007.

```
template <typename Key, typename Node,
   template <typename,typename> class Ordering>
struct LookupInNode;

template <typename Key,
   template <typename,typename> class Ordering>
struct LookupInNode<Key, NullType, Ordering>
{
  typedef NullType result;
};

template <typename Key, typename Value,
   typename Left, typename Right,
   template <typename,typename> class Ordering>
struct LookupInNode<Key, TreeNode<Pair<Key,Value>,
   Left,Right>, Ordering>
{
  typedef Value result;
};

template <typename Key, typename SplitKey,
   typename SplitValue, typename Left,
   typename Right, template <typename,typename>
   class Ordering>
struct LookupInNode<Key, TreeNode<Pair<SplitKey,
   SplitValue>,Left,Right>, Ordering>
{
  // Note that the = case is handled by the
  // partial specialization above.
  typedef typename Select<Ordering<Key,SplitKey>::
     less,LookupInNode<Key,Left,Ordering>,
     LookupInNode<Key,Right,Ordering>
     >::result::result result;
};

template <typename Key, typename Tree>
struct Lookup;

template <typename Key, typename Root,
   template <typename,typename> class Ordering>
struct Lookup<Key, Tree<Root,Ordering> >
{
  typedef typename LookupInNode<Key,Root,
     SubsidiaryOrdering<Ordering>::Eval>::
     result result;
};
```

Listing 7

```
template <template <typename,typename> class
Ordering> struct SubsidiaryOrdering;

template <>
struct SubsidiaryOrdering<KeyOrder>
{
  template <typename x, typename y>
  struct Eval
  {
    enum { less = Order<x,y>::less };
    enum { greater = Order<x,y>::greater };
  };
};
```

Listing 8

Java Protocol Handlers

Roger Orr demonstrates the use of Java's URL handling to make code independent of the source of data.

Introduction

In today's programming environment data can be sourced from a variety of locations, using a range of protocols. In many cases the actual source of the data is irrelevant to the application; when this is the case then being able to abstract details of location away from the code means that we can process data from a variety of different places by simply changing a configuration string.

So, for example, the configuration for the common Java logging suite log4j [log4j] can be provided as easily from a local file on the hard disk as from an Internet web site without any changes being required to the application code.

As another example, an overnight batch process might take data via ftp from a remote server, but be more easily tested by running against a sample disk file containing a known dataset.

The standard method of describing such abstract locations for data is through a URL (Universal Resource Locator) – the most common example of these being web site address such as http://www.accu.org.

Java comes with built-in support for URLs, most obviously through the **`java.net.URL`** class.

A simple example of using the URL class

Listing 1 is a trivial Java program which makes use of the **`URL`** class to read data in a location agnostic manner.

```
package howzatt;

public class Example {
  public static void main( String[] args ) {
    for ( String uri : args ) {
      read( uri );
    }
  }

  public static void read( String uri ) {
    try {
      java.net.URL url = new java.net.URL( uri );
      java.io.InputStream is = url.openStream();
      int ch;
      while ( ( ch = is.read() ) != -1 ) {
        System.out.print( (char)ch );
      }
      is.close();
    }
    catch ( java.io.IOException ex ) {
      System.err.println( "Error reading " +
        uri + ": " + ex );
    }
  }
}
```

Listing 1

This program can be run against:

- a local file using an argument like **`file:example.txt`**
- a remote file using an argument like **`file://server/path/file`**
- a Web site using an argument like **`http://www.accu.org/`** (subject to any restrictions by network firewalls or proxies).

URLs – or 'what's in a name?'

The syntax of a URL is defined by RFC 2396 [RFC2396] which, confusingly perhaps, uses the term URI (Universal Resource Identifier) rather than URL. The difference between the two is that a URI is more general; it can describe resources that aren't locations, for example urn:isbn:978-0-470-84674-2 which is the ISBN number of a book. In practice, however, the distinction between the two terms is often blurred. There is a fuller discussion of this issue on the w3c site [W3C – URI].

Each identifier before the first colon in a URI name defines a 'scheme' and schemes such as **`http`** and **`ftp`** are globally recognised as standard. The full list of official schemes is held by the Internet Assigned Numbers Authority [IANA].

Java has support for URIs and URLs through the **`java.net.URI`** and **`java.net.URL`** classes. Additionally, Java is supplied with inbuilt support for a number of different schemes, as a minimum support for the following is guaranteed: **`http`**, **`https`**, **`ftp`**, **`file`**, and **`jar`**.

Note that Java refers to these schemes as 'protocols' although, for example, processing an **`http`** URL involves two protocols – DNS to resolve the host name and HTTP to access the data.

Although the five standard protocols are often adequate, there are sometimes cases where access is required to other data sources. Often the location of this data can be described using the URI syntax but it may not be an 'official' URI scheme.

For example, data might be obtainable using **`scp`** (secure file copy) and the obvious URI of **`scp://user@host/path/file`** could be used to represent the location of a file on some remote host. Or again, data may be supplied in a zip file or some other compressed format and you want to be able to access the data uncompressed from within your program.

Fortunately Java allows us to supply our own protocol handlers to extend the set of supported schemes.

There are existing extensions to the Java protocol handlers provided by various sites on the Internet and supporting various protocols; one such example is Hansa [Hansa]. If your requirements are for support of a well-known protocol you may be able to find a pre-written protocol handler.

Roger Orr Roger has been programming for over 20 years, most recently in C++ and Java for various investment banks in Canary Wharf and the City. He joined ACCU in 1999 and the BSI C++ panel in 2002. He may be contacted at rogero@howzatt.demon.co.uk

For stand-alone applications the easiest way to register your new protocol is to define the system property used by the URL class

However, there may be times when you want to implement a protocol handler yourself – whether for an unsupported official scheme or for a proprietary one.

Extending Java's URL handling

Java supplies a standard mechanism for extending the supported protocol schemes, which is described in brief in the documentation for the `URL` class. The process consists of three parts:

- writing a class, derived from `java.net.URLStreamHandler`, that knows how to open a connection to URLs of the new scheme
- writing a class derived from `java.net.URLConnection`, to access data from these connections
- associating the stream handler class with the protocol name

The first two steps will obviously depend heavily on the specifics of the protocol being supported, and may involve such actions as opening network connections or invoking external programs. I'll illustrate the process with a very simple example that uses the 'quote of the day' service to make the general principle clear without requiring too many protocol-specific details.

The final step involves plugging your new classes into the processing the `URL` class uses when it comes across a protocol for the first time. The `URL` class attempts to create an instance of the correct `URLStreamHandler` class in the following order:

- If a factory has been registered with the `URL` class then the `createURLStreamHandler` method of the factory is called with the protocol name.
- If there is no factory, or the factory does not recognise the protocol, then Java looks for the system property `java.protocol.handler.pkgs` which is a `|` delimited list of packages. For each package it tries to load the class `<package>.<protocol>.Handler`, which, if present, must be the `URLStreamHandler` for the given protocol.
- Failing this the system default package is searched for a handler in the same way.

For stand-alone applications the easiest way to register your new protocol is to define the system property used by the `URL` class; so let's see how this might work.

A 'quote of the day' handler

A standard Internet service is supported, on many operating systems, on port 17. This service simply returns a random quote whenever a TCP/IP connection is made to it. If this service is available on your machine, you can see it at work using telnet. Here is a Windows example:

```
C:> telnet localhost qotd
"We want a few mad people now. See where the sane
ones have landed us!"
George Bernard Shaw (1856-1950)
Connection to host lost.
```

```
package howzatt.qotd;

public class Handler
extends java.net.URLStreamHandler {
  protected java.net.URLConnection
  openConnection(java.net.URL u)
  throws java.io.IOException {
    return new QotdConnection( u );
  }
}
```

Listing 2

If this attempt fails, you might need to start the service (or connect to another machine that does offer the qotd service). On Windows it is one of the 'Simple TCP/IP Services'.

In order to access this service from my example program at the start of the article I need a URL syntax, so I've picked the simple format:

`qotd://hostname`.

Since we are using an unofficial scheme there are several alternative ways of encoding the data as a URI.

Listing 2 contains example code for a simple stream handler for the qotd protocol and the actual connection handling code itself is in Listing 3.

```
package howzatt.qotd;

public class QotdConnection
extends java.net.URLConnection {

  private static final int QOTD = 17;
  private java.net.Socket socket;

  public QotdConnection( java.net.URL u ) {
    super( u );
  }

  public void connect()
  throws java.io.IOException {
    final String host = getURL().getHost();
    socket = new java.net.Socket( host, QOTD );
    connected = true;
  }

  public java.io.InputStream getInputStream()
  throws java.io.IOException {
    if ( ! connected )
      connect();
    return socket.getInputStream();
  }
}
```

Listing 3

Now if we compile these two additional classes, we can use the `qotd` protocol with the example program shown earlier like this:

```
java -Djava.protocol.handler.pkgs=howzatt howzatt.
Example qotd://localhost
```

If all is well we get a quote displayed – we have transparently extended our simple application to acquire data from a different source.

Problems with protocol handlers

In my experience the biggest problem with extending Java's protocol handlers is with the registration process. Writing the code to handle the specific protocol is a fairly clear task, it requires a decision about the URI syntax to be used for and the code written for the particular connection type.

The registration problem is harder because of two design issues.

- The factory registration is inextensible
- The class loader used by the `URL` class cannot be changed

As a mentioned earlier, one way of registering your `URLStreamHandler` class with the `URL` class is to provide a factory object. Unfortunately this mechanism is somewhat inflexible; specifically the `setURLStreamHandlerFactory` method can be called at most once in a given Java Virtual Machine.

This may be a valid restriction for a small Java application but it becomes hard to manage when two different parts of the application, possibly written by unrelated teams, each wish to register a factory for their own protocol with the `URL` class.

However, even leaving this problem aside, the factory approach requires the application code to register the factory explicitly which makes it hard to add new protocols to existing programs. This is what we did earlier to the example program, and is one of the most powerful aspects of Java's protocol handler support.

On the other hand, using the `protocol.Handler` convention can be problematic because of the way Java class loaders work.

When a new protocol is detected by the `URL` class it tries to load the appropiate handler class but using the class loader that was used to load the `URL` class itself.

For a stand-alone application this does not usually present a problem, but where the Java code is running inside a web service or as an applet it is normal for user-supplied code to be loaded by a different class loader than the core Java classes.

In these cases, any protocol handler class supplied in the user code will not be found by the system class loader used to load the `java.net.URL` class.

In these cases it also may not be as simple to externally configure the system property used by the `URL` class and the `System.setProperty` method can be used at runtime to add additional packages. Note however that this approach might be barred by the security manager and care must also be taken to ensure that any existing packages defined by this system property are retained. See Listing 4.

```
public static void register() {
  final String packageName =
     Handler.class.getPackage().getName();
  final String pkg = packageName.substring(
     0, packageName.lastIndexOf(  '.' ) );
  final String protocolPathProp =
     "java.protocol.handler.pkgs";

  String uriHandlers = System.getProperty(
     protocolPathProp, "" );
  if ( uriHandlers.indexOf( pkg ) == -1 ) {
    if ( uriHandlers.length() != 0 )
      uriHandlers += "|";
    uriHandlers += pkg;
    System.setProperty( protocolPathProp,
       uriHandlers );
  }
}
```

Listing 4

Alternative approaches

Given the problems with registration, other approaches can be taken. One is to jettison the Java URL and provide a different abstraction; this seems to be the approach favoured by the Apache 'Commons Virtual File System', which retains the use of the URI syntax but provides an alternative method of access the data using a `FileSystemManager` class.

The weakness with such an approach is that it does not of itself support handling of additional protocols when using existing code that uses the `java.net.URL` class internally to connect to a URL.

Another approach is to use the factory registration, but to provide a factory class that itself supports registration of multiple different stream handlers using different names.

This approach supports code using the `java.net.URL` class but it does require a registration call for each protocol and so hence changes are needed to an application before it can make use of the new URLs. However the approach gets around the problems discussed above with multiple class loaders since the factory is loaded by the user code class loader rather than by the class loader for the `URL` class.

Restrictions

The Java protocol handlers are not suitable for every situation. There are two main reasons for this.

- The URL abstraction may hide too much detail of the underlying data representation. For example, processing might require file-system specific methods, or be intolerant of network latency.
- Not all resources are easily described by a URI, and not all protocols fit into the `URLConnection` model. Security can be a particular problem here since the usual way of including a username/password into a URL uses plain text which is obviously rather insecure.

Conclusion

The location abstraction provided by the URL notation makes it possible to write programs that can transparently access data from a wide variety of different places.

There is a parallel with the way that Unix treats 'everything like a file' – even access to system information. This common view of data means that simple tools may have wide applicability. The same principle applies with the use of URLs in Java – the abstraction can make programs able to process a wide range of data from a variety of sources without needing explicit coding.

Java provides a relatively simple mechanism to add new protocols to your applications and hence widen the range of locations for sourcing data.

There is a great deal of power in this approach; sadly the specific details of registering with the `URL` class are not very flexible but in most cases there are various techniques to work around the limitations. ■

References

[log4j] 'Apache log4j', http://logging.apache.org/log4j/

[RFC2396] 'Uniform Resource Identifiers (URI): Generic Syntax', http://www.ietf.org/rfc/rfc2396.txt

[W3C-URI] 'URIs, URLs, and URNs: Clarifications and Recommendations', http://www.w3.org/TR/uri-clarification/

[IANA] 'Uniform Resource Identifer (URI) Schemes', http://www.iana.org/assignments/uri-schemes.html

[Hansa] 'Project Hansa', http://wiki.ops4j.org/dokuwiki/doku.php?id=hansa:hansa

[VFS] 'Commons Virtual File System', http://commons.apache.org/vfs/

Upgrading Legacy Software in Data Communications Systems

Changing operational software is a risky business. Omar Bashir offers a case study in matching risk with reward.

Introduction

The article 'Trouble with TCP' in the CVu issue of December 2006 [CVu06] highlighted issues in implementing (near) real-time point to multipoint communications over TCP. The author had highlighted the constraint of minimum code rewrite that most developers face while upgrading legacy systems to resolve various issues. This constraint had prevented the author from applying any of the alternative solutions that he mentioned at the end of his article. This article describes a similar legacy system that faced serious performance issues when traffic on the system increased from moderate to high levels. A solution that did not involve legacy software rewrite was attempted to resolve these performance issues. Various aspects of this solution are explained here.

The legacy system

The legacy system was a monitoring application aggregating data from a number of sensors in a data multiplexer (MUX) and displaying this data in real time on a number of workstations on a LAN. The sensors are connected to the MUX via dedicated and secure communication links. In addition to the workstations, the LAN also hosts a database server logging the data from the sensors for historical analysis and an application server on which various near real time trend analysis applications are executed. Figure 1 shows the physical topology of the system.

Operator workstations, database server and application server (referred to as clients) established connections to the MUX over TCP upon boot up. Copies of every message received by the MUX from the field sensors were transmitted over each established connection. At a modest message arrival rate of 100 messages a second and with the resulting LAN packet size of 256 bytes, communicating these messages to 5 clients only will result in traffic rates of over 1 Mbps. Therefore, this system operated satisfactorily at low message rates for a few clients. Only by increasing the number of clients, the data rate on the LAN increased appreciably. This coupled with higher incoming message rates (due to either increased field events being monitored or increased number of field sensors) could cause congestion either due to network limitations or limitations of the MUX platform software (e.g., buffer sizes).

As often happens with useful applications, a few months after its induction, the number of operator workstations was increased and so were the number of field sensors. This resulted in a noticeable delay between monitoring events at the sensors and those events being displayed on the operator workstations.

Omar Bashir Omar first experienced software development about 15 years ago writing programs for automatic test systems while working as an avionics system engineer. He has ever since written software for telecommunications, logistics and financial applications and enjoys integrating machines and proceses.

He can be reached at obashir@yahoo.com

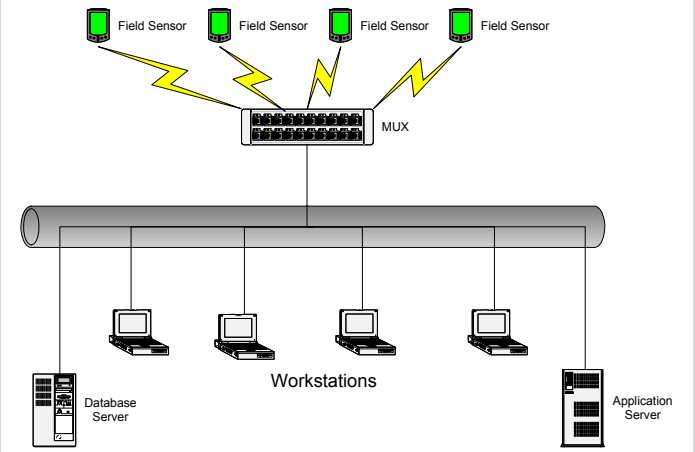

Figure 1

It was decided to consider changing the transport protocol to UDP with minimal (preferably no) change to the existing software. The remaining article discusses various factors that were considered while deciding the change in the transport protocol for the application and the resulting solution that required no change in the existing software.

Considering a UDP-based solution

TCP, because of the reliability guarantees it offers, is generally considered suitable for loss sensitive applications whereas UDP is connectionless and inherently unreliable. However, compared to UDP, TCP's reliability is associated with overheads in its implementation and operation. This difference in the fundamental principles of these transport protocols has given rise to some practices in network programming. For example, TCP is considered suitable for reliable and sequenced but non real-time delivery of application data. UDP, on the other hand, has been the protocol of choice for (near) real-time applications that are insensitive to a degree of loss.

The choice of transport protocol, however, needs to be based on a comprehensive analysis of application requirements. For instance, the performance of TCP in comparison to that of UDP has been argued in high throughput and loss sensitive environments. The performance of UDP-based applications is expected to drop due to the retransmission of packets lost because of the absence of flow control [Snader00]. On the other hand, broadcast and multicast communications is a feature inherently supported by the connectionless nature of UDP whereas this feature has to be engineered at the application layer in TCP-based applications as multiple unicast transmissions. As discussed above, the feasibility of the latter approach is questionable at higher throughputs coupled with increasing number of receivers as the transmitter iterates over a list of receivers practically replicating every packet for every receiver.

UDP's susceptibility to loss can cause failure in applications sensitive to packet loss. Even if every packet is not required for the real-time operation, it may be necessary to record all data communicated between different

fragmented packets are only reassembled at the destination and the loss of even a single fragment results in considering the entire packet being lost

TCP and UDP

TCP (Transmission Control Protocol) and UDP (User Datagram Protocol) form the transport layer protocols (layer 4 protocols) for the Internet Protocol suite. Although the primary objective of both these protocols is the same, i.e. to allow distributed and networked application components to communicate with each other via message passing without concerning themselves with the characteristics of the network(s) connecting them, the characteristics of these two protocols are very different making them suitable for applications with specific communication requirements.

The network layer protocol of the Internet Protocol suite is called the Internet Protocol (IP) and it is a best-effort protocol. Therefore, it makes the best possible effort (without any guarantees) to deliver packets containing user messages to the destination. It treats each packet differently, determines the route these packets need to take to reach the destinations and possibly even break them into smaller packets if required. If a packet is fragmented, it is reassembled only at the destination and not at the intermediate nodes (routers). Packets may, therefore, get lost, duplicated or arrive out of order at the destination.

TCP sits over IP and provides message validation facilities. It ensures that messages sent by the remote application layer processes are received without errors and in the sequence in which they are transmitted. Therefore, TCP handles retransmission of lost and corrupted packets, discards duplicates and rearranges the received packets to reconstruct the message stream. Furthermore, TCP manages flow of the data streams to avoid and alleviate congestion.

In order to perform these operations, a connection needs to be established between source and destination applications. In order to establish a connection, one process listens for a connection request where as the other attempts the connection. A listener is usually the process that provides services to the connecting process and is referred to as a server. The connector is, therefore, referred to as a client. However, logically it is possible but not common for clients to be listeners and servers to connect to listening clients and provide information when the required information is available.

As multiple networked applications may execute concurrently over the same host, they need to be identified using a unique identifier called the port number. Combination of the port number of an application and the IP address of the host on which it executes uniquely identifies that application on the Internet. This combination is referred to as socket endpoint. A TCP connection is an association between two endpoints, one identifying the client and the other identifying the server. Therefore, a connected socket endpoint pair uniquely identifies a TCP connection.

Messages are transmitted between the connected applications as a stream of sequenced bytes. Therefore, for most implementations of TCP, an application may provide TCP with discrete units of data at the transmitting end but TCP returns to the application at the receiving end an unfragmented stream of bytes. If messages are to be retrieved as discrete units of data then message boundaries need to be explicitly specified and then looked for within the received byte stream. For example, if lines of text are transmitted using TCP and the receiving application needs to receive and process these lines individually, the received byte stream needs to contain line feeds that the receiving application can look for to determine line boundaries within the stream. A common method of delineating binary messages in a TCP byte stream is to insert a message header of a fixed size containing the message length. The receiver initially reads the header (as it is of fixed size), determines the size of the remaining message and then read it. This process is repeated for the subsequent messages.

IP creates packets containing portions of these streams and transmits these packets to the destination. TCP at the receiving end attempts to recreate the stream using the sequence numbers providing by TCP at the transmitting end. Receiving TCP acknowledges the receipt of the last sequenced byte received indicating the receipt of all bytes up to the sequence number. Absence of an acknowledgment at the transmitting TCP indicates a packet loss, necessitating the retransmission of all packets from the last successful acknowledgment.

The connection-oriented nature of TCP does not allow multicast or broadcast communication of messages. This is accomplished by UDP. UDP is a connectionless protocol, i.e. the receiver of messages can be sent messages arbitrarily as long as transmitters know the socket addresses of the receiver without establishing a connection prior to transmitting these messages. UDP is a very simple protocol that provides the same best effort delivery of messages that IP offers to packets at the network layer. Therefore, each message (or datagram) at UDP is treated independently; it can be lost, duplicated and delivered out of sequence. For this reason, delineation of messages with UDP is not required. Multiple receiver processes can bind to a multicast or the broadcast IP address and a common port to simultaneously receive broadcast or multicast messages. Usually, applications using UDP need to provide mechanisms to deal with packet losses, duplications and out or sequence arrival. However, care needs to exercised while developing such applications so as not to re-engineer TCP over UDP.

components of the system for historical analysis. A degree of reliability can be engineered over UDP at the application layer but over-engineering leading to re-engineering TCP at the application layer over UDP is usually discouraged (e.g., [Snader00]).

A simple approach to mitigate packet losses over UDP is defined for facsimile transmissions over UDP in the ITU-T.38 standard [ITU-T.38]. This approach employs redundancy to overcome packet losses. Each UDP datagram encapsulates, along with the current Internet fax packet, a predetermined number of previously transmitted Internet fax packets. For example, with Internet fax packet n5, packets n4, n3 and n2 are also transmitted and with Internet fax packet n6, packets n5, n4, and n3 are also transmitted. Thus, even if packets n4, n3 and n2 are lost, n5 packet allows the recreation of the missing stream. Implementations of redundancy-based schemes to mitigate packet losses should restrict the overall packet size to lower than the smallest MTU (Maximum Transmission Unit) along the packet's path otherwise fragmentation of the packet may occur thereby increasing the packet loss probability. This is because fragmented packets are only reassembled at the destination and the loss of even a single

the communication protocol was required to be changed from TCP to UDP with preferably no software change in the existing software suite

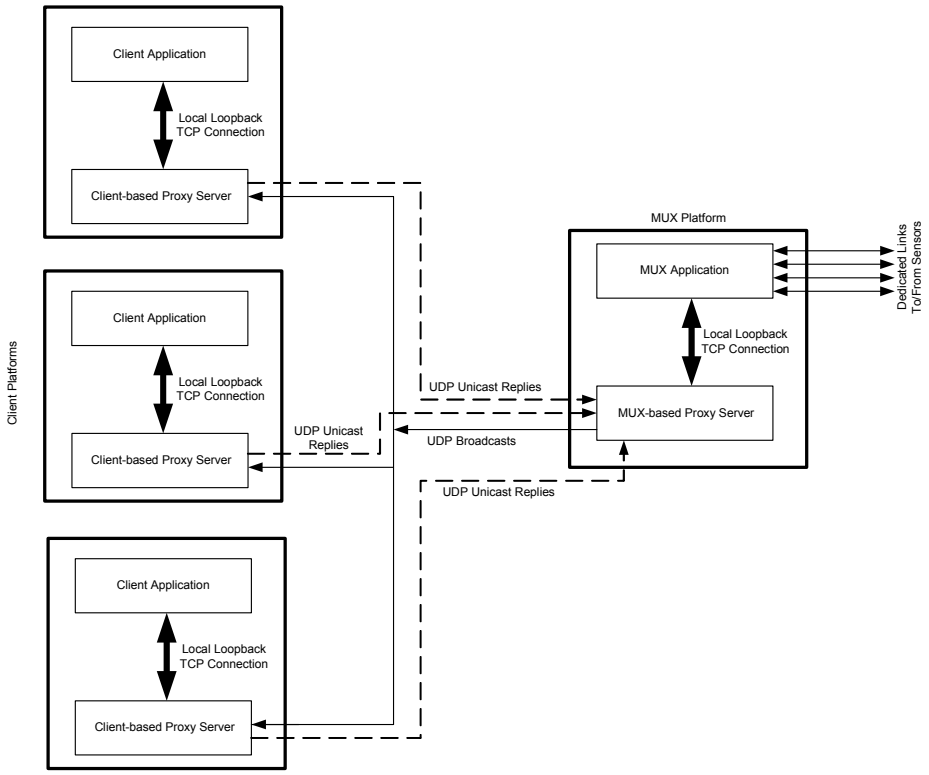

Figure 2

fragment results in considering the entire packet being lost [RFC0791]. UDP may also be appropriate for tunneling non-Internet protocols that provide end-to-end reliability, e.g. facsimile communication [ITU-T.30].

Solution architecture and implementation

In the system described earlier, the communication protocol was required to be changed from TCP to UDP with preferably no software change in the existing software suite. This was accomplished by implementing proxy servers on both the MUX and the client platforms. The MUX-based proxy server executes on the MUX platform and acts as a client application to the MUX. In this case, the MUX has only one client and that is the local proxy server, which connects to the MUX over TCP via local loopback. Upon receiving data from the MUX application, the proxy server broadcasts the data on the LAN over UDP to be picked up by the client-based proxy servers.

The client-based proxy servers are bound to a broadcast port and receive the data broadcast by the MUX-based proxy server. These client-based proxy servers act as MUX applications for the client applications executing on their respective platforms. Client applications are reconfigured to connect to their local proxy servers over TCP via local loopback. Client applications thus consider their local proxy servers to be the MUX application. The data received by the client-based proxy servers via the broadcast ports is transmitted to the respective local client applications via a TCP connection over the local loopback. Figure 2 shows the high-level architecture of the proposed solution.

Client applications may need to communicate with the sensors in the field via the MUX. Data from the client applications is communicated to their respective local proxy servers over TCP connections via local loopback. The local client-based proxy servers transmit these messages over UDP to the MUX-based proxy server, which passes it to the MUX application over the TCP connection via the local loopback. The MUX application can determine from the message the sensor to which this message is to be transmitted and this message is then transmitted over the relevant dedicated link to the sensor.

Because only one client (i.e., the MUX-based proxy server) connects to the MUX application, only one copy of each incoming message from the sensors is transmitted over the local loopback to the proxy server. As these messages are broadcast over the LAN, only one copy of each message ever exists over the network.

Detailed description of proxy servers

Figure 3 shows the high-level block diagram of the proxy server. For full-duplex communication, the proxy server needs four threads. Two of the

a proxy server may be required to transform messages it relays between the systems it connects

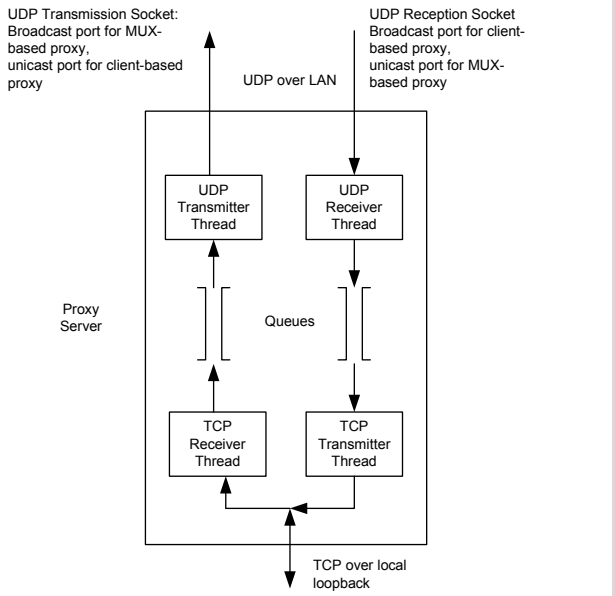

Figure 3

threads are used for communicating over two UDP sockets and the remaining two are used for communicating over a single TCP socket. For the MUX-based proxy server, a UDP socket is used to transmit to a broadcast port and a socket bound to a unicast port is used for reception. For client-based proxy servers, a socket bound to a broadcast port is used for reception and messages for the MUX-based server are transmitted by another UDP socket to a port open at the MUX-based proxy server for reception.

The TCP Receiver Thread communicates with the UDP Transmitter Thread using a queue. Similarly, the UDP Receiver Thread communicates with the TCP Transmitter Thread over another queue. These queue objects are instantiations of a wrapper around an STL queue providing thread safety and blocking read operation using mutexes and condition variables provided by the operating system.

A relatively more detailed class diagram of the proxy server is shown in figure 4. Communicator Thread is the abstract base class for the classes that implement thread objects in the proxy server. Each communicator thread uses a Thread Safe Queue object to either write received messages to (in case of objects of the Receiver Thread sub-class) or read received messages from (in case of objects of the Transmitter Thread sub-class).

A proxy server may be required to transform messages it relays between the systems it connects. This transformation is performed in the Transmitter Thread class using an appropriate concrete extension of the Transformer abstract class. Transformation is performed in the Transmitter Thread as it may be more time consuming for large messages or messages requiring significant processing during transformation. If performed in the Receiver Thread, it may cause packet losses. It is possible to apply different transformations to different types of messages or messages containing specific content by implementing transformers using the Strategy design pattern [Gamma95].

Transmitter and Receiver Thread objects use objects of concrete sub-classes of the `Communicator` abstract base class to receive and transmit data over sockets. Objects of `UdpCommunicator` class are used to communicate over UDP sockets whereas objects of `TcpCommunicator` class are used to communicate over TCP sockets. Messages received over TCP need to be delineated from the input stream. `TcpCommunicator` objects use objects of sub-classes of `AbstractDelineator` to perform delineation of the input stream. These classes implement application specific stream delineation logic.

`TcpCommunicator` objects are created by `Initiator` or `Acceptor` factory classes which are derived from the `Connector` abstract base class. This is a variation of the ABSTRACT FACTORY design pattern [Gamma95]. An `Acceptor` object accepts connection requests from TCP clients and returns a `TcpCommunicator` object encapsulating the connected socket. Similarly, a TCP client uses an `Initiator` object to initiate a connection to a TCP server. `Initiator` also returns a `TcpCommunicator` object encapsulating the connected socket.

Concluding remarks

TCP is usually the transport protocol of choice in data communication applications that are loss sensitive. However, TCP's inherent inability to handle point to multipoint communication can severely restrict system scalability resulting in latencies that may be unacceptable even in systems with relatively relaxed delay sensitivities. Increase in message sizes, number of clients or number of messages to be broadcast per unit time can result in increased delays as well as resource utilization at the servers.

UDP's ability to broadcast and multicast packets over a LAN can provide the required scalability. However, UDP being a best effort protocol is not considered suitable for loss sensitive applications. Some application level enhancements can provide a degree of resilience against packet losses. However, migrating an application from TCP to UDP may require a significant rewrite. A proxy server based approach is presented here that allows TCP based point to multi-point applications to be migrated to UDP without rewriting the existing application. Each host on the network executes a proxy server, which communicates over UDP with other proxy servers. Proxy servers communicate with the application components on the local host via TCP over local loopback.

Using proxy servers helped in future product growth as further developments could be phased appropriately. The design mentioned above was generalized and formalized as the Active Transceiver architectural pattern for data communication systems [Bashir03]. The next phase of development resulted in a MUX software that communicated over UDP without the MUX-based proxy server while not changing any of the client applications. In the subsequent phases, each different client application was modified to communicate over UDP, in addition to other application specific enhancements. ■

References

[Easterbrook06] Mark Easterbrook, 'Trouble with TCP', *CVu*, December 2006, pp 8-9.

[Snader00] Jon Snader, *Effective TCP/IP Programming: 44 Tips to Improve Your Network Programs*, Addison Wesley, 2000.

[ITU-T.38] International Telecommunication Union, 'ITU-T T.38 Procedures for real-time Group 3 facsimile communication over IP Networks', 1998.

[RFC0791] J Postel, 'RFC-0791 Internet Protocol', Internet Engineering Task Force (IETF), September 1981.

[ITU-T.30] International Telecommunication Union, 'ITU-T T.30 Procedures of document facsimile transmission in the general switched telephone network', 1997.

[Gamma95] Erich Gamma, Richard Helm, Ralph Johnson, John Valissides, Design patterns : elements of reusable object-oriented software, Addison Wesley, 1995.

[Bashir03] Omar Bashir, Mubashir Hayat, Active Transceiver Design Pattern for Data Communication Applications, IEEE 7th International Multi-topic Conference 2003 (INMIC 2003), Islamabad, Pakistan, November 2003.

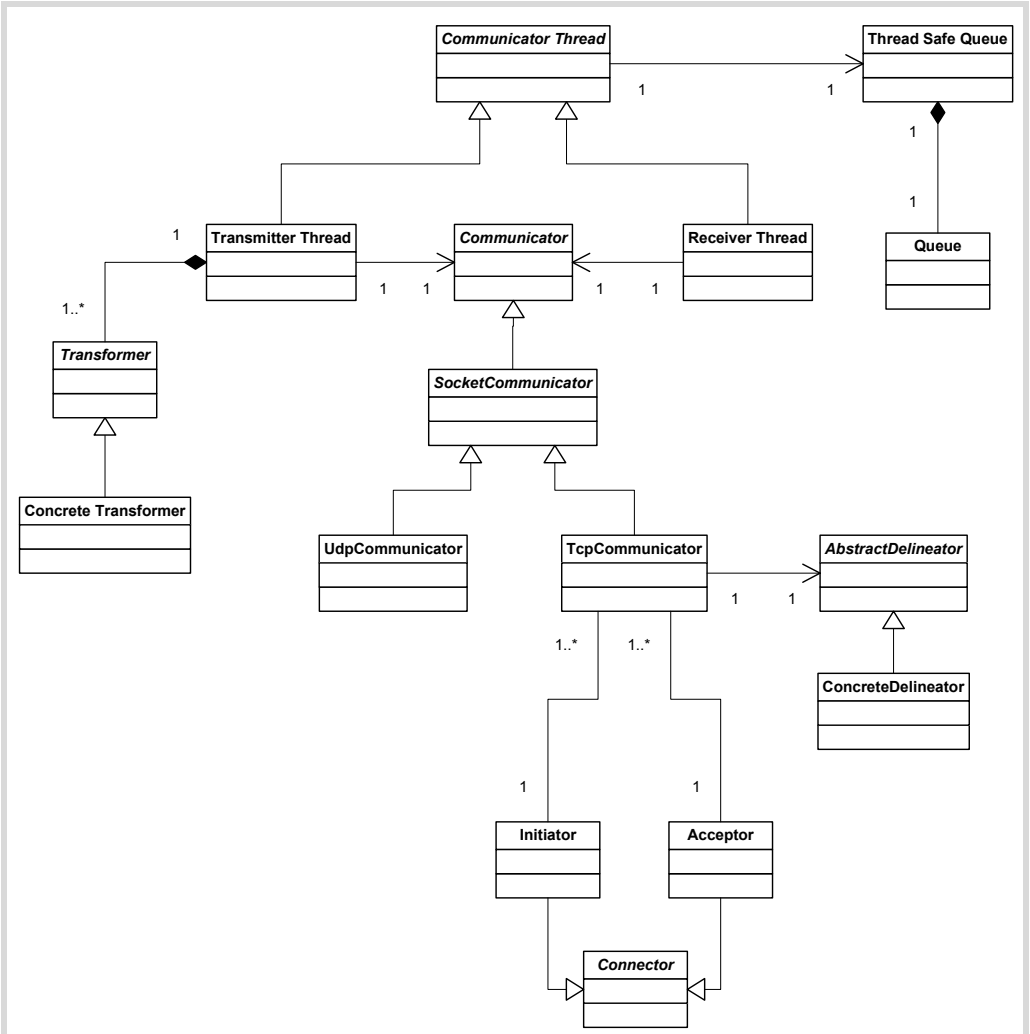

Figure 4

Creating Awareness Exposing Problems

One of the good things about presenting at the ACCU conference is what you learn there. Allan Kelly reviews the results of his last conference talk.

Background

At the last three ACCU conferences I have given semi-interactive sessions. Anyone who has attended these sessions will know the form: I present some slides and talk about some ideas for the first half. As I talk people respond and contribute their thoughts and ideas. I note these ideas down on a flip chart and after about 30–40 minutes nobody really cares about my slides because we are having a really good discussion. The audience learns something and more importantly so do I.

What might be less well know is that I take these flip charts home – or at least good digital pictures of them. Then sometime after the conference I write up notes. In 2005 this was just a web-page with 31 bullets points to capture the ideas . The 2006 presentation, 'Changing your organization' led to a six-page write up on thoughts and ideas .

This year I've repeated the process, only this time I decided to expand the write up with some notes on the presentation and publish it here in Overload (unfortunately this rather grander ambition also accounts for the lateness). I hope those of you who attended the presentation will find this reminder useful and I hope those who were not able to attend will find something of interest here. Although the audience suggested most of the ideas presented here I have added my own notes and thoughts to expand on the ideas.

Creating awareness and exposing problems

In many ways this session was a continuation from the previous two years. The unifying theme is that in order to develop software better we need to learn and our organizations need to change. It is not enough to learn, we need to act on that learning. For learning to be meaningful it must lead to action and create change.

The first step in this process needs to be exposing the problems we face and the opportunities available, and creating awareness about these issues. Hence this session.

Of course we all want to live a better life: write new code, have less bugs, get paid more, have a bigger house but things get in the way, and overcoming these obstacles is difficult. This is hard enough when it is just us but when it is our team or our entire company it is more and more difficult. It is far easier to shut up, stay quiet and accept things the way they are; but if you were such a person you probably wouldn't have joined the ACCU, probably wouldn't read Overload and certainly wouldn't attend the conference!

Overcoming these obstacles and making life better requires effort. All too often the effort required stops us from changing things. So how can we overcome these obstacles?

As it happens some of these blocks are in our own heads. These blocks are relative easy of overcome because we are in complete control of them. All we have to do is recognise the block and choose to change ourselves. Nothing is stopping us except ourselves.

Other obstacles are more difficult. In order to introduce a change into our team or organization then we need other people to agree to the change and help out. There are basically three approaches to this.

Option 1: Tell them what to do

This is the Hollywood model. Arnold Schwarzenegger parachutes into the development department to save the project. Armed only with an Uzi sub-machine gun, several hand-grenades and a hybrid-Hummer H2 he orders

> You, you and you, code up the user interface – no bugs or the blonde gets it
>
> You there, take 3 programmers, secure the bug tracking system and eliminate all bugs.
>
> I'll deal with the customers... If I'm not back in 30 minutes call in an air-strike.

Well there are a few problems here:

- *Do you know enough to tell them?* Before you can tell someone what to do you need to know what to tell them. For big, complex, problems this isn't a trivial matter.
- *Will they do what you say?* Maybe people you work with recognise your good ideas, understand them perfectly and act on them exactly as you describe. But since I've never ever encountered such an environment I'm guessing you don't work in such a place either.
- *Will they understand your commands?* Most developers know how ambiguous requirements can be: the same is true with instructions to change. When people don't do what you expect it may they simply don't understand your request or don't see the world the way you do. Try not to jump to the assumption that they are deliberately trying to be difficult and obstructing your efforts.
- *Do you need to check on them?* Unless you are confident that people understand what you expect, and you trust them to do what you want then you will spend a lot of time checking on them. And if you find you are spending a lot of time checking on them then ask yourself: *Am I really trusting them?*

  If you need to check on people then it is going to take a lot of your time so you will be less productive. Plus to this the people you are checking up on are unlikely to enjoy the checks and will feel less trusted. Should you leave, or stop checking, things might just go back the way they were.
- *What about motivation?* People who just do what they are told are usually a lot less motivated than those who are involved with the decision making process. Motivated people are more productive so we need to find a way to keep people's motivation while we change the way they work.

Allan Kelly Allan served his apprenticeship developing software for financial, communication and utility systems. He is now a consultant and interim manager who specialises in advising and helping the most challenged development teams deliver and improve. His first book, Changing Software Development, is published by John Wiley and Sons early next year. Allan can be reached at allan@allankelly.net

Faced with fear and threats people are quite likely to stick their head in the sand, put problems out of their mind and carry on as before

This option is predicated on the assumption that you have authority to tell people what to do; and that the people you are telling will accept your authority. On balance this probably isn't a good option.

Option 2: Scare them into changing

Faced with serious problems like 'the company is going bust' or threats like 'our new zero tolerance programme means we will shoot the next developer who writes a bug', it is quite possible you can persuade people to change. After all who wouldn't?

However it is also quite possible that your scare tactics will have the opposite effect. Faced with fear and threats people are quite likely to stick their head in the sand, put problems out of their mind and carry on as before. You may even make things worse, individuals may well adopt behaviours which shield them from any threat at the expense of addressing it. For example, threatened with a company failure people may decide to find a new job; or faced with penalties for writing bugs they may simply stop writing any code.

You might just scare people into doing things differently and it might just work – great! However, what happens next time you need a change? Will the scare tactics work a second time? A third?

To complicate things further, success can breed complacency and it can make things hard to change in future. When people have success doing things one way they are likely to want to repeat that success. Persuading them to try something different risks losing the past success. It seems counter intuitive to be told that future success depends on dropping existing practices that have brought success.

So option two isn't reliable either.

Option 3: Help others to change

Suppose most people probably feel the way you do: they would like the world to be better too. These people also have ideas on how to improve things. However for them, unlike you, the effort is too much. The solution therefore is to help them, reduce the effort needed and overcome the blocks.

The first thing to do is recognise the blocks, understand where effort is needed. When you recognise blockages and share the understanding it becomes easier to remove the blocks – many hands make for light work.

This option starts to sound a lot better. And this is what the rest of the presentation and this report looks at.

Recognising obstacles is half the problem, but it is not enough for you to see the obstacles. Other people must see the same problems, opportunities and obstacles that you do. If they can't then maybe you see the wrong ones. In order to agree on the issues you need to share the understanding.

Ideas from the group

That was the introduction. Next I asked the audience for ideas. What follows are the best suggestions from the group, with some elaboration from me.

Retrospectives

It was hardly surprising that one of the first ideas suggested from the audience was 'hold project retrospectives'. Now retrospectives are a great idea, they can work wonderfully. Project retrospectives are not confined to the software development domain; under the name 'After action reviews' they are used by the US Army, Marines Corp and the British National Health service. (I'm not saying every NHS operation or military battle is subject to a review but they do hold some.)

Many books on process improvement suggest project retrospectives and there are two excellent books on the subject. Norm Kerth's *Project Retrospectives* is orientated towards reviews at the end of long projects. Diane Larson and Esther Derby published *Agile Retrospectives* last year, which discusses the use of retrospectives in Agile teams and over a shorter period (several weeks as opposed to several years in Kerth's.)

The conference audience came up with a few more ideas, some of which are covered in these books and some not:

- Writing things down can help record them; projects could have a log book that records the events in a project as it unfolds.
- Retrospectives should be held regularly and acted upon within the project.
- Choice of retrospective facilitator is important. The facilitator needs to be able to facilitate the retrospective, i.e. encourage people to speak and explore the issues raised.

  They also need to be apart from the retrospective, if the facilitators have a lot to say themselves then they aren't going to be able as effective in getting others to speak. Secondly, if the facilitator is in the management team of a project their presence and control of the retrospective may inhibit the free discussion.
- Retrospectives can become dominated by one or more "big mouths" – someone who uses the forum as an opportunity to talk and talk and talk. A good facilitator will know how to manage these people and give others a chance to speak, in the extreme you may want to exclude these people from the retrospective.
- Retrospectives need safety: people can't speak openly and discuss problems unless they feel they are in a safe environment.

The group also identified and discussed several 'failure modes' of retrospectives. By far the most common failure of retrospectives is that they simply don't happen. Everyone seems to agree that 'retrospectives are good' but many people are reluctant to schedule the time for them to happen.

A second failure mode occurs when a retrospective happens but it is too late for it to make any difference. In many companies project teams are broken up after a project finishes, this makes it harder to hold a retrospective and harder to apply the lessons of the retrospective. Similarly, when project teams are staffed with a lot of contractors and consultants who leave at the end of the project it is both difficult to learn and apply the lessons. One solution to these problems is to put

In most organizations the people who do the work know what is wrong, they know what needs fixing and they know who are the effective people and who is free-loading

retrospectives inside the project rather than at the end – following the Derby and Larson model rather than the Kerth one.

When retrospectives do happen the biggest problem seems to be obtaining buy-in from everyone concerned and acting on the conclusions. Lack of buy-in manifests itself in two ways, firstly getting the right people (i.e. everyone involved with the project) to actually attend and contribute to the session. Secondly getting buy-in from those in authority to act on the conclusions.

However acting on the conclusions is not a management problem alone. Developers, testers, analysts and others need to act too when the retrospective suggests changes. It is not enough to blame management for a lack of change.

Personally I would add one more item to the problems identified by the group. This is: time. Almost without exception whenever I have scheduled a retrospective people have been taken aback by the amount of time I have allowed. For a project of six months I might schedule a whole afternoon, for a six-week project I might want two hours. If a project has not held a retrospective before I would allow more time.

This looks like a long time to people with busy schedules but if you want to understand what happened, understand the causes and devise meaningful solutions it is barely enough time. (And that is not counting write up time after the session is finished.) Looked at in terms of the overall project these time periods are not long: a three-developer project lasting six weeks represented over 90 days of effort, probably more when you add in project management and software testing. Is two hours really too much time to spend on learning the lessons of such a project?

Pub conversations / Safe and trusting environment

One idea that came up in the discussion was the 'pub' or 'water cooler conversation'. These are the conversations that occur out-of-band from the office work. The fact that these conversations occur isn't surprising however they are symptomatic of an environment where people cannot talk about these things in the office or to their managers.

In most organizations the people who do the work know what is wrong, they know what needs fixing and they know who are the effective people and who is free-loading. However all too often this knowledge is not available to managers. People talk about these things but they talk outside the office, in the pub, the coffee shop or at the water cooler in the corner of the office.

This information is not available to managers for a variety of reasons. Firstly managers need to make time to join these conversations and listen. Sometimes this requires creating a forum (like a retrospective) where the discussion can happen. The manager could simply join the guys in the pub but it's better if such discussions can occur in a more sober environment.

A second block to making these conversations more useful is trust. People will not discuss some matters unless they trust the person they are talking with. Without trust people will not feel safe enough to hold open conversations. Managers need to create an environment were people feel safe and can trust one another. Without trust and safety people will doubt others' motives and guard what they say.

This is not to say that every pub conversation needs bringing inside the organization and acting on. Many such conversations are riddled with personal opinions, biases and large quantities of alcohol. There may also be legal limits in both what people say and what they choose to hear. For example, if a company is likely to be taken over in the next few days people may not be able to speak freely about the future. Neither are managers at liberty to discuss individuals who may have resigned or be under disciplinary action.

It was suggested that one way of promoting safety is to keep people informed. Again there may be legal limits here, particularly if a company is publicly listed. Organizations do need to communicate with themselves, without communication different groups within organizations may become detached. This is bad enough when it is between business units, e.g. sales and development, it is worse when it occurs between the company's leaders and the workers.

Publish problem and solution

Identifying and solving a problem is not necessarily the end of the story. In an organization that is trying to learn and improve widely – especially if the organization is large – it is reasonable to try and share your new knowledge. Even if your organization is small and ad hoc there is still benefit from writing up what you have achieved:

- You might learn something new as a result of writing up the solution
- You have a record for yourself
- You could share your solution with others outside the organization, say in the pages of Overload.

Give someone else your idea

Sometimes when you see a problem or an opportunity you are not in a position to act on it yourself. Or perhaps you can act on it but you need others to help you. In these circumstances don't be scared to give your idea away. Mention it to others, spread the idea and make sure those who can do something about it know.

However, you cannot be possessive. When you hear someone else talking about your idea, don't rush to say 'I told you that' or 'That was my idea'. Smile to yourself but let others own your idea too. After all it is more important that the idea is acted upon than it is for the credit to be apportioned.

Over time people will notice you are the source of good ideas, and they will remember it was you who first pointed out what is now obvious. You have to play the long game.

One more point from my personal observation. If you have suggested or warned of something and it comes to pass then avoid the temptation to say 'I told you so' or 'That is what I have been saying'. Saying this once in a while is fine, but it can be tiring when someone constantly tells you they foresaw events. So before you say 'I told you this two months ago', ask yourself if you really need to point out how right you were.

use good metrics to help understand and visualise the system

Metrics

Metrics can be a useful way to expose problems. However there are a number of problems with metrics that the group was quick to identify. Finding a good metric is difficult: ideally they need to be easy to capture (and/or calculate) and they need to be easy to understand. If it isn't easy to get a metric they will fall into disuse, and if they are difficult to understand only a few people will be able to use them.

Do people work to a metric? Numerous studies show that people will change their performance when a metric is introduced. As a result the metric may improve but some other aspect of the system may change.

The British tabloid press have labelled this condition 'targetitis' when it occurs in relation to hospital targets. For example, a hospital may be set a target of discharging patients within a certain time period, say two days. In order to meet the target the hospital may discharge all patients after two days and immediately re-admit them in. This would meet the target number but not the target intention. More troublesome would be a patient discharged prematurely who then develops complications and needs to be re-admitted. In such a case the well intentioned target becomes dangerous.

Another variation on this is known as 'gaming the system'. This occurs when an individual stands to gain from some outcome. The individual knows the rules of the game and attempts to use the rules to achieve the outcome even at the expense of the overall outcome. For example, say a developer offered a bonus for delivering on schedule. With this incentive they may ignore requests for changes, refuse to acknowledge bugs or cut functionality.

The problem is not so much to do with metrics but targets. Using a metric to monitor and understand a system is one thing but targeting a numeric value for a metric can change the behaviour of the system. This is known as Goodhart's law after the British economist who first identified this effect.

So if you can find a good metric do not make it into a target! *Measurements are not targets*.

One suggestion from the group was to use good metrics to help understand and visualise the system. Burn-down charts are good example of this. These show how the development system is performing without creating targets.

Another suggestion was to use the metrics to promote competition between teams. This might create indirect targets but if the competition is kept good humoured and reasonable it could promote learning between the teams as they compete to do better.

Show responsibility

Sometimes individuals can play a very direct role in uncovering problems simply by taking responsibility and accepting their own mistakes. It is natural that when one makes a mistake that creates problems for others we don't want to talk about it. We might even try to avoid it or hide it. This can make life more difficult for others and sets a bad example. If we are serious about improving our organizations, exposing problems and creating awareness we need to set an example and own up to our mistakes.

The flip side of this is to be fair to people who admit to mistakes. We shouldn't single them out for criticism or behave detrimentally to them. If someone has admitted a mistake then they show be respected and rewarded. Again there is a need for a safe and trusting environment.

Showing responsibility also means you seek to understand other people and their thinking. Before you rush to brand someone's actions 'a mistake' and expect them to admit it, consider if it was a mistake by their norms. What we see as a mistake, or wrong, might simply be a different way of working. This is particularly true when co-workers come from a different background, perhaps different technology, type or organisation or even country. Things are not always so black and white.

Where is the Promised Land?

When we are aiming to improve our environment and solve problems it helps if we know where we are heading. We, as individuals who read Overload, might know exactly where we are heading: we are heading to a bug free land where code is delivered on schedule, testing is fully automated and we all work just 40 hours a week. However do others see this land? Or do they see your efforts as pointless? Just something else to make their lives hard?

Explaining where we are heading is only the first stage. You also need to explain why we are heading there and get everyone to agree on where you are going. I believe Steve Jobs once said:

> *It doesn't matter how we get to San Francisco as long as we all agree we are going to San Francisco. The problem is when someone secretly wants to go to San Diego.*

Once everyone agrees to go the same place it becomes much easier to do the right thing.

Finally

Hopefully by the time you read this I'll have my original presentation posted on my website at http://www.allankelly.net.

I would like to thank my audience for their many good suggestions. I would have liked to spend more time on some of the ideas in order to get down to the detail of how they can be implemented but there is never enough time to do everything.

In the end each of us has to find what works for us, in our own environment. Knowing what other people do can inspire us but it can never give us all the answers we need. Some things we have to discover for ourselves. ■