overload 91

JUNE 2009 £3

On Management: The Business Analyst

We look at the role of the Business Analyst in healthy software development

FastFormat, Part 3

We continue the dissection of Matthew Wilson's C++ formatting library

Floating Point Fun And Frolics

Numbers are remarkably hard to handle correctly. Here are some (floating) points to consider.

Complexity, Requirements and Modules

We take lessons from Christopher Alexander and learn how to organise code modules along their natural complexity boundaries

A magazine of ACCU

ISSN 1354-3172

OVERLOAD 91

June 2009

ISSN 1354-3172

Editor

Ric Parkin
overload@accu.org

Advisors

Phil Bass
phil@stoneymanor.demon.co.uk

Richard Blundell
richard.blundell@gmail.com

Simon Farnsworth
simon@farnz.co.uk

Matthew Jones
m@badcrumble.net

Alistair McDonald
alistair@inrevo.com

Roger Orr
rogero@howzatt.demon.co.uk

Simon Sebright
simon.sebright@ubs.com

Paul Thomas
pthomas@spongelava.com

Anthony Williams
anthony.ajw@gmail.com

Advertising enquiries

ads@accu.org

Cover art and design

Pete Goodliffe
pete@goodliffe.net

Copy deadlines

All articles intended for publication in Overload 92 should be submitted by 1st July 2009 and for Overload 93 by 1st September 2009.

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

4 Floating Point Fun and Frolics
Frances Buontempo investigates the dark corners of floating point numbers.

9 On Management: The Business Analyst Role
Allan Kelly continues his look at roles in software management.

12 Complexity, Requirements and Models
Rafael Jay considers the problem of excess software complexity.

16 An Introduction to FastFormat (Part 3): Solving Real Problems, Quickly
Matthew Wilson shows FastFormat in practice.

26 The Model Student: The Enigma Challenge
Richard Harris poses a historical problem.

29 Boiler Plating Database Resource Cleanup (Part 2)
Paul Grenyer finds a better way to clean up in Java.

36 ACCU 2009
Giovanni Asproni provides a report.

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.

Copyrights and Trade Marks

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

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

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

A Good Craftsman Knows His Tools

Are you using the right ones?
Ric Parkin looks in his toolbox...

At the highly enjoyable ACCU conference this year, I took part in a patterns hatching workshop. This took the form of a brainstorming round where people came up with names of possible patterns (and not just design patterns – organisational ones were also prominent), followed by further rounds where other people tried to fill in the details, for example what problem it solves, what sort of circumstance it applies in, and the consequences of the solution, both positive and negative (always good to remind people that a pattern is not a silver bullet and always has effects which lead to further problems to solve).

But one stuck in my mind as being oddly unsatisfactory: ‘Use Tools’. We really struggled to fill in the details for this one. Initially it sounds uncontroversial: who wouldn’t use tools after all! But in a way that’s its weakness – if it’s universally applicable, then it's not actually a very interesting pattern. Also, we couldn’t really find anything much to say about the details – what sort of tools, what problem do they solve etc., as we could think of so many different possible things, but they were all to do with more specific ideas. I eventually came to the conclusion that it is just too broad a idea to encapsulate in a single pattern form, although there are lots of patterns within it.

Man is a tool-making animal – Benjamin Frankin

But it did get me thinking about tools. Until relatively recently it was considered that a defining trait of Humans was to be the use of tools. But recent discoveries that not only many primates [BBC] but many other species use tools [Species], and even some crows are true tool makers including passing on new uses culturally [Crow]. But it is the sheer scale of tool use amongst humans that still impresses. Combine this with the ability to communicate, which helps to spread new ideas quickly, and it is easy to understand how an otherwise unremarkable species can dominate its environment.

So what is a tool? A useful definition is that they are something used intentionally to magnify our ability to do something. It could be as simple as using a picked up stone to break another in two, or using a branch as a lever to move a bigger rock than you could otherwise handle. More advanced is to deliberately shape objects to work as better tools, such as chipping a stone to create a sharp edge to be used when butchering, or sharpening the branch to become a spear. Increasing our strength is one possible improvement – better accuracy and repeatability are others, and all help to make best use of otherwise limited resources.

So what sort of tools do we use in the world of software development? We could start with the hardware – a modern computer is a general purpose computing tool. The really clever and important part is in fact the software that turns it into a specialized computing tool. A quick look at the history of computing shows that some of the really momentous breakthroughs were moving away from custom built to general purpose tools, which could be more easily built and then customised as needed. A prime early example was the Jacquard Loom [Jacquard], which made making complex textiles easier and more flexible, because instead of having a different machine for each pattern (or relying on a slow error-prone human) it was a general machine that allowed the pattern made to be programmed and changed easily. A similar significant step was the creation at Bletchley Park of Colossus which was the first true programmable computer [CodesAndCiphers]. Before that, the machines used to crack codes were hardwired for a particular code – if the rotors on an Enigma machine changed, you had to physically rebuild your machines.

But Colossus and similar early machines were still pretty hard to program, using technologies such as a plug-board and cables to wire up your program. A real step forward was to create tools that helped you write programs. There are many different types, but we’ll start with a major one that is the very bedrock of a computer – the Operating System. This provides the platform upon which every other tool rests, and so it influences hugely your choice of further tools. This could be a major restriction until you consider cross-platform development – with that you can choose a development platform distinct from the target platform your project will eventually run on. This is essential where the target platform isn’t itself capable of hosting your development tools, which will be true of many embedded devices, mobile platforms, and games consoles. There can still be some lock in: if a vendor only produces the tools to run on a particular operating system, then you don’t have much choice.

The operating system will often come with many simple utilities, which can be useful as a box of Lego bricks to build custom tools. Unix systems are notably rich in command shells that run script languages, and for small, specialised programs that take input, process it in some way, and pipe it on to the next program, which can make it easy to write the small utilities that ease day to day tasks. As an example, a few years ago I wrote a simple Integration Build Server. It was a simple bash script that kept an eye on the source code repository; if it

Ric Parkin has been programming professionally for around 20 years, mostly in C++, for a range of companies from tiny startups to international corporations. Since joining ACCU in 2000, he’s left a trail of new members behind him. He can be contacted at ric.parkin@gmail.com.

saw the root time stamp change it would wait a until it hadn't changed again for 10 minutes (this was to allow people to check in a series of changes one by one); it then checked out the latest code, ran the build script (which itself tied together many different tools), captured the output and mailed the results to the development team. The script was only around a hundred lines long, and ran on both Unix and Windows (using a port of the unix utilities), but saved the team a lot of pain from broken builds and reliability.

The next obvious category of tools would be the platform for the actual product itself. This could be the major infrastructure such as database, web server, or content management system. These will often be tightly aligned to the choice of operating system, and often the choice is imposed on us by business or client needs.

The choice of programming language is another major choice, although one we can more often influence to some extent. It could be an interpreted language such as python, a web toolkit such as Ruby On Rails, or a traditional compiled language such as C++, or more commonly a combination of several different technologies. The choice will often be constrained by non-technical influences such as fashion for the latest cool thing, prejudice for or against certain tools, and the existing areas of expertise of the deciding team, and the ability to hire developers with knowledge in that area.

Then there are the tools that help you craft your product. The text editors, code browsers, unit test systems, debuggers, profilers, graphics editors, dialog layout tools and so on.

And finally there are the various ancillary tools that help you manage your team and the development processes – email, messaging tools, meeting calendars, UML editors, the humble word processor for documentation, source code repository tools.

All these tools come together to help you build your product, which is itself likely to be a tool for your clients to do their own work.

If the only tool you have is a hammer, you tend to see every problem as a nail – Abraham Maslow

We all come with a variety of experiences and history of using various tools. But beware: this can lead us into the trap of only considering the tools we are already familiar with for our next project, and rejecting those we have no knowledge of, or only poor experiences whether or not it was the fault of the tool itself. While choosing the unfamiliar is itself a risk, being closed to other options could lead us to use completely inappropriate tools, which can vastly increase our development costs and risks.

References

[A.P.R.I.L.F.O-O.L] http://accu.org/var/uploads/journals/overload90.pdf – contents

[Bickerstaff] http://en.wikipedia.org/wiki/Isaac_Bickerstaff

[BBC] http://news.bbc.co.uk/1/hi/sci/tech/4296606.stm

[CodesAndCiphers] http://www.codesandciphers.org.uk/lorenz/colossus.htm

[Crow] http://en.wikipedia.org/wiki/New_Caledonian_Crow

[Jacquard] http://en.wikipedia.org/wiki/Jacquard_loom

[Species] http://en.wikipedia.org/wiki/Category:Tool-using_species

[Tool] http://en.wikipedia.org/wiki/Tool

Corrections and Clarifications

Readers of Overload 90 may have noticed that an article – 'Array Problems: Range Iteration Logic for Object-Oriented Languages' [A.P.R.I.L.F.O-O.L] was advertised as on page 37 of a 36 page magazine. We have Swiftly apologised to the author [Bickerstaff] for this one-past-the-end error, and will endeavour to avoid a repeat during the next 12 months.

Floating Point Fun and Frolics

Representing numbers in computers is a non-trivial problem. Frances Buontempo finds using them is hard, too.

It is unworthy of excellent persons to lose hours like slaves in the labour of calculation.
Gottfried Wilhelm von Leibniz

This article is a summary of a recent discussion on *accu-general* concerning floating point numbers. First, we look at how numbers are represented on computers, starting with whole numbers and moving on to real numbers. Next, we consider comparing numbers stored in floating point representation, noting pitfalls and common mistakes along the way. The third section investigates performing calculations with floating point numbers, noting the significance the order of calculations makes and gives warning of clever tricks that might not actually be all that clever. Mathematical proofs of the accuracy of the algorithms and tricks presented are not given: the purpose of this article is simply to give a taster of what is possible and what should be avoided. Working with floating point numbers is hard, frequently mistake ridden, but with care and attention can be done sensibly.

Representations

Let us begin with non-negative integers or whole numbers. These can be easily represented in binary. Using 32 bits, we can represent numbers between 0 and 2^{32}-1. For signed numbers, we could use a bit to represent the sign, and the remaining bits to represent a magnitude. Alternatively integers can be encoded using the more common two's complement. A non-negative number is stored as the binary representation, while a negative number -y is stored as 2^{32}-y. This is equivalent to switching the bits and adding one. For example, let us see how -1 is represented. Flipping the bits on 1 gives

```
11111110
```

Adding 1 gives the representation of -1 as

```
11111111
```

As a sanity check, we should find if we add 1 we get zero.

```
 11111111
+00000001
 00000000    (ignoring the overflow carry bit)
```

This representation allows subtraction to be performed using just the hardware for addition. Overflow can occur for either approach, since only a fixed number of bits are available. This is dealt with in different ways by different programming languages. Only a finite set of integers can be represented: work is required to represent arbitrarily large or small numbers.

Frances Buontempo has an undergraduate degree in Maths + Philosophy, an MSc in Pure Maths and a PhD in technically Chem Eng, but mainly programming and learning about AI and data mining. She has been a programmer for over 10 years professionally and learnt to program by reading the manual for her Dad's BBC model B machine. She can be reached at frances.buontempo@gmail.com

Rational numbers, or fractions, can be represented symbolically by storing the numerator and denominator, though arbitrarily large integers will still need to be represented to cover all the rational numbers. This symbolic representation requires several extra calculations for basic arithmetic. It does not allow exact representation of irrational or transcendental numbers, such as $\sqrt{2}$ or π. The obvious alternatives are fixed or floating point encodings, allowing the representation of real numbers with a degree of precision.

Fixed point representation uses one bit for the sign, a field of a fixed number of bits before the decimal point, and a fixed length field for the fractional part after the point. If 32 bits are used, with one bit for the sign, 8 bits before and 23 bits after the decimal point, we can only represent numbers between 2^{-23} and 2^8– 1. In addition, numbers can now underflow (anything less than 2^{-23}), as well as overflow (anything greater than or equal to 2^8), and the numbers we can represent are not dense (in lay person's terms they have gaps in-between, unlike rational or real numbers: whichever two you think of, no matter how close, you can always find another one in between) and so we will have rounding errors to consider.

An alternative is floating point, based on scientific notation. Any real number, x, is written as a sign followed by a mantissa or significand, S, and a magnitude or exponent, E, of a base, usually 2 or 10. Nowadays, computers commonly use base 2. For example:

$$x = \pm S . 10^E$$

The decimal point 'floats' as the exponent changes, hence the name. Under-and overflow still occur if a fixed number of bits are used for the exponent and mantissa, however a greater range of numbers can now be represented than with fixed point.

This general sketch of floating point representation gives much scope for discussion. How many digits should the significand and the exponent be? How do you represent the sign of the exponent? 'Anarchy, among floating-point arithmetics implemented in software on microprocessors, impelled Dr. Robert Stewart to convene meetings in an attempt to reach a consensus under the aegis of the IEEE. Thus was IEEE p754 born.' [IEEE98]

IEEE defines several representations of floating point data, along with negative zero, denormal or subnormal numbers, infinities and not-a-numbers (NaNs) [IEEE87]. It also covers rounding algorithms and exception flags. As with any floating point representation, a finite number consists of three parts: a sign, a significand (or mantissa), and an exponent. This triple is encoded as a bit string, whose length varies depending on which representation is being used. The IEEE standard gives 'three basic binary floating-point formats in lengths of 32, 64, and 128 bits, and two basic decimal floating-point formats in lengths of 64 and 128 bits'. [IEEE87] A conformant programming environment provides one or more of these formats. The base may be either 2 or 10, and each limits the possible ranges for S, giving the precision, and E, giving minimum and maximum possible exponents, emin and emax.

Of course, the floating point representation of a specific number will not be unique without a convention: + 1.0 x 2^0 == + 0.1 x 2^1. In order to give a unique encoding for floating point number, the significand S is chosen

Mapping a real number to a floating point representation may incur rounding, which signals an inexact event

so that either E = emin, or S>=1. If S<1, the number is denormal or subnormal, having reduced precision but providing gradual underflow. They cause special issues for rounding: the error bounds on underflow differ from those on normal numbers. For normal numbers using base 2 the rounding error will be at most one unit in the last place, ulp, $2^{-(p-1)}$, where p is the precision. This does not hold for denormal numbers. Without these numbers, the gap between zero and the smallest representable number is larger than the gap between two successive small floating point numbers. More details on their controversial introduction are given in [IEEE98].

The signed bit is 0 for positive and 1 for negative for all representations. For a 32-bit format, 8 bits are used for the exponent, ranging between -126 and 127. This is stored as a biased exponent adding 127 to get a value in the range 1 to 254. For double precision the bias is 1023. For single precision, exponents of zero and 255 have special meanings, leaving 23 bits. If we are representing normalised numbers in binary, they all start $1.x_1x_2x_3$..., so there is no point in storing the 1. If the exponent is all zero, we are dealing with denormalised numbers, $0.x_1x_2$... .

Let us consider the bit pattern of some single precision binary numbers.

```
S EEEEEEEE FFFFFFFFFFFFFFFFFFFFFFF
0 1......8 9..................... 31
```

Note first, we have two zeros:

```
0 00000000 00000000000000000000000 = 0
1 00000000 00000000000000000000000 = -0
```

As mentioned, all the ones for the exponent have special meanings:

```
0 11111111 00000000000000000000000 = Infinity
1 11111111 00000000000000000000000 = -Infinity
0 11111111 00000100000000000000000 = NaN
1 11111111 00100010001001010101010 = NaN
```

Otherwise, for non-zero exponents we have

```
  0 10000001 10100000000000000000000
= +1 * 2^(129-127) * 1.101 = 6.5
  1 10000001 10100000000000000000000
= -1 * 2^(129-127) * 1.101 = -6.5
```

And finally we have denormal numbers, such as

```
  0 00000000 00000000000000000000001
= +1 * 2^(-126) * 0.00000000000000000000001
= 2^(-149)
```

For w exponent bits, the biased exponent allows every integer between 1 and 2^{w-2} inclusive to encode normal numbers, the reserved value 0 is used to encode ±0 and subnormal numbers, and the reserved value 2^{w-1} to encode ±Inf and NaNs (quiet and signalling). A quiet NaN is the result of an invalid operation, such as 0/0 or a operation with a NaN for one operand, and silently propagates through a series of calculations. Whenever a NaN is created from non-NaN operands, an **INVALID** flag is raised. A signalling NaN must be explicitly created and raises a floating point exception when accessed which could be picked up by a trap handler. They can be used to detect uninitialised variables. As we have seen, all the bits in the biased exponent field of a binary NaN are 1. In addition, a NaN will contain diagnostic information in its 'payload', giving a family of NaNs. Note that NaNs 'compare unordered with everything, including itself'. [IEEE87]. Infs are constructed as the limits of arithmetic with real numbers, allowing operations with them to be exact rather than signalling an exception event. As an aside, note that IEEE allows decimal floating point numbers to have more than one encoding: numerically equal 'cohorts' are identified. Different cohorts can then be chosen for different operands. The specifics of Infs and NaNs differ, but they are still easily represented.

Mapping a real number to a floating point representation may incur rounding, which signals an inexact event. There are five IEEE rounding options. It can be round to the nearest number: for 'ties', such as 4.5, rounding is either to the nearest even number, here 4.0, which is the default, or away from zero: for 4.5 this will be 5.0. The former, also known as bankers' rounding, makes sense, as it will sometimes round down, and sometimes round up, so in theory all the small differences will come out in the wash. Consider 0.5 + 1.5. This should be 2.0. If we round each number away from zero to the nearest whole number, we calculate 1.0 + 2.0 = 3.0. If bankers' rounding is employed, the sum is 0.0 + 2.0 = 2.0, giving the correct answer. Other forms of round can be set: round towards zero, or truncation, and round towards plus or minus infinity.

Exceptions are error events, rather than language specific exceptions. There are five possible events: underflow, overflow, division by zero, inexact and invalid operations, such as square root of a negative number. By default, a flag will be set to indicate if such an event has occurred, though this behaviour can be overridden for example by installing a trap handler. The specifics are platform dependent. Once a flag has been set, it is 'sticky', that is it remains set until explicitly cleared.

Many gotchas can occur with floating points. Rounding errors occur before any calculation for some numbers when more bits are required than are available in the representation. Similarly underflow or overflow can cause problems. This includes comparison and calculations. Unintuitive things occur, since floating point operations are neither commutative nor associative. Furthermore, some compiler optimisers assume algebraic properties hold and make mistakes. What do you think **x1** and **x2** will be in your favourite C++ compiler?

```
double v = 1E308;
double x1 = (v * v) / v;
double temp = (v * v);
double x2 = temp/v;
```

See [Monniaux08] for further fun and frolics your compiler, platform and optimiser can cause.

Comparisons

Let's start with a simple example of what can go wrong when floating point numbers are compared. Consider:

```
bool test = 0.10000000000000001
    == 0.1000000000000000;
```

What is the value of test? What would a 10 year old tell you? Your computer will tell you the value on the left hand side is in fact equal to the

I have seen code out there in the wild that just checks if two numbers are within epsilon of one another. This will (almost) never be what is required.

one on the right, whether you try this in C, C++, Python or Haskell. This happens because of the underlying IEEE representation of the number. Simply put, 'Squeezing infinitely many real numbers into a finite number of bits requires an approximate representation'. [IEEE87] So, how should floating point numbers be compared?

Numbers could be compared to a pre-specified number of decimal places [ACCU09]:

```
bool equal(double x, double y, double places)
{
  return floor(x * pow(10.0, places) + 0.5) ==
     floor(y * pow(10.0,places) + 0.5);
}
```

Multiplication by 10^{places} moves the required number of digits to the left of the decimal place. Addition of 0.5 ensures that numbers ending with the digit 5 are rounded up. This is a common trick. This comparison will be unsatisfactory in general. How many decimal places are required? What happens for really big or really small numbers? Comparison to a specific number of decimal places is probably not sensible. Instead, what is required is a function that detects whether two numbers are close. The decimal place comparison above can be adopted to check if two numbers are equal to a given number of significant figures. Alternatively, the relative difference can be employed.

Consider the function [ACCU09]:

```
bool equal(double x, double y, double precision)
{
  return fabs(x-y) < precision;
}
```

This certainly checks if two numbers are within a given range of one another. However, it will be hard to use, as the precision is an absolute difference, and must vary with the two numbers being compared. What value is suitable for

`equal(1000000000001,1000000000002, ???)`?

What about

`equal(0.000000000001,0.000000000002, ???)`?

Epsilon may spring to mind, for example in C++, `std::numeric_limits<double>::epsilon()`. This is the smallest number that can be added to 1.0 to obtain a number that doesn't compare equal to 1.0 The C++ standard defines 'Machine epsilon: the difference between 1 and the least value greater than 1 that is representable' [C++03]. I have seen code out there in the wild that just checks if two numbers are within epsilon of one another. This will (almost) never be what is required. For bigger or smaller numbers, we require a multiple of epsilon, or our chosen precision, giving the relative difference. [ACCU09]

```
bool equal(double x, double y, double precision)
{
  return fabs(x-y) < fabs(x)*precision;
}
```

This relative error approach still raises questions. How do we decide whether to multiply by `x` or `y`?

I suspect rounding errors, particularly ones in production systems initiating support calls at 3am, cause programmers to first start thinking about comparing floating point numbers. Numbers are required to be close enough, allowing for computational rounding. We can check if two numbers are within the worst-case difference that can be caused by rounding errors. For one rounding error, resulting from one calculation, the absolute difference will be less than `fabs(x)*epsilon`, as used in the relative error above. For n IEEE floating point operations, a seemingly sensible comparison is therefore [ACCU09]

```
fabs(x-y) <= double(n)*fabs(x)*epsilon;
```

However, what happens if `x` is zero? We then check the difference is less than or equal to zero, which is not the intention. It is safer to use [ACCU09]

```
fabs(x-y) / sqrt(x*x + y*y + epsilon*epsilon)
  < n * epsilon;
```

Calculations

Comparison of floating point numbers is just the beginning of the matter: the order of calculations can also make a difference. Consider three ways of summing some floating point numbers. See Figure 1.

This gives `sum1` as 1000000000.0000000, and `sum2` and `sum3` as 1000000000.0000001. Addition is neither associative nor commutative in

```
double sum1 = 1000000000.0 + 0.00000001 + 0.00000001 + 0.00000001 + 0.00000001 + 0.00000001 +
              0.00000001 + 0.00000001 + 0.00000001 + 0.00000001 + 0.00000001 + 0.00000001 +
              0.00000001;

double sum2 = 1000000000.0 + (0.00000001 + 0.00000001 + 0.00000001 + 0.00000001 + 0.00000001 +
              0.00000001 + 0.00000001 + 0.00000001 + 0.00000001 + 0.00000001 + 0.00000001 +
              0.00000001);

double sum3 = 0.00000001 + 0.00000001 + 0.00000001 + 0.00000001 + 0.00000001 + 0.00000001 +
              0.00000001 + 0.00000001 + 0.00000001 + 0.00000001 + 0.00000001 + 0.00000001 +
              1000000000.0;
```

Figure 1

Catastrophic cancellation occurs if two numbers are close and are the result of previous calculations, so have inaccuracies in the lower digits

floating point arithmetic, giving a very strange number system. Sorting the numbers to be summed so that the smallest numbers are aggregated first will be more accurate, however this will be time consuming. A better approach is the Kahan Summation Formula [Goldberg91]. This takes into account the lower order bits, avoiding some rounding errors as it runs. For the sum of an array of n floating point numbers, x, Kahan's summation is calculated as follows:

```
double sum = x[0];
double small_bits = 0.0;
for( int j = 1; j < n; ++j )
{
  double y = x[j] + small_bits;
  double temp = sum + y;
  small_bits = y - (temp - sum);
  sum = temp;
}
```

This illustrates one approach to calculations – compensating for lower order bits that would otherwise get lost. On the first step round the loop, for `sum1`, `sum = 1000000000.0`, so the smaller `0.00000001` has got lost. However, it is remembered in the `small_bits`, and will be taken into account in the final result.

We have seen how addition can cause problems: so can subtraction. Catastrophic cancellation occurs if two numbers are close and are the result of previous calculations, so have inaccuracies in the lower digits. Most of the accurate digits may cancel as a result of subtraction, leaving behind cruft in the lower order digits. On the other hand, if the operands are exactly known rather than calculated their difference is guaranteed to have a very small relative error, which is referred to as benign cancellation. Consider, the mathematically identical formulae $x^2 - y^2$ and $(x + y)(x - y)$. Though they are identical algebraically, they may not be computed to be equal. [Goldberg91] demonstrates why the right hand side is usually a more appropriate computational approach. Intuitively, when x and y are close, $x^2 - y^2$ will be very small, giving a potentially large relative error. However, although the expression $(x - y)(x + y)$ does not cause a catastrophic cancellation, it is slightly less accurate than $x^2 - y^2$ if x is much smaller than y or vice versa . In this case, $(x - y)(x + y)$ has three rounding errors, but $x^2 - y^2$ has only two since the rounding error committed when computing the smaller of x^2 and y^2 does not affect the final subtraction. [Goldberg91]

Sometimes it is tempting to rearrange a mathematical formula for greater efficiency, but mathematically equivalent formulae have different 'stability properties' [Highham96]. Consider calculating the variance of an array of numbers.

```
variance = 1/(n-1) sum( x-mean(x) )
```

where

```
mean = 1/n(sum(x))
```

This involves two passes through the numbers, once to calculate the mean and a second time to calculate the difference from the mean. Mathematically this is identical to the sum of squares method [Cook]

```
variance = 1/n(n-1) (n.sum(xi^2) – (sum(xi)^2))
```

This second formulation can lead to difficulties. For example, the variance can become negative if the numbers are large with comparatively small differences. Try it with 1.0e09 + 4, 1.0e09 + 8, 1.0e09 + 13. This gives about -156.77. This will cause trouble if we wish to calculate the standard deviation, that is the square root of the variance.

A better approach is Welford's method [Welford62], which is an 'on-line algorithm'–- that is, it provides the new result in one step for a new value of x, rather than having to recalculate everything from the start, so will give us the efficiency we require but without the potential instability. This is achieved via a recurrence relationship or iteration, which calculates the new value from the previous value. If we have the current mean for $k – 1$ numbers, for a new value x

```
mean = previous_mean + (x - previous_mean)/k
```

The variance can then be calculated from

```
sum_variance = previous_sum_variance +
               (x - previous_mean)*(x-mean)
```

as `variance = sum_variance / k;`

This version is both a one-pass algorithm and does not suffer from the same potential rounding problems as the sum of squares method. Any formula can be expressed in more than one way. It is worth considering the rounding errors that can occur and looking for ways to rearrange the order of the calculation to avoid them. As a quick finger in the air test of a formula, compare the output for the same sequence of numbers presented in different orders: if they differ you probably have a rounding error. Then try different rearrangements try for your formula, with the inputs presented in different orders. This can be a quick way to detect rounding errors. Another approach is to try one formula with different rounding modes set. Furthermore, your optimiser can also wreak havoc with your carefully laid plans. Changing your optimisation settings, re-running your program and examining the results can reveal some fun and frolics.

Other representations

Each of the rounding errors considered so far stem from the way real numbers are represented on a machine. Many real numbers cannot be represented precisely in binary using a fixed number of bytes. There are many ways to represent the real numbers on a computer. These include binary-coded decimal (BCD) [Wikipedia], p-adic numbers [Horspool78] and IEEE floating point standard [IEEE87].

BCD typically uses four bits to represent each digit. For example, 127 would be encoded as

```
0001 0010 0111
```

It takes more space than floating point representation, but decimals like 1/10 which recur in binary (0.0001100110011...) can be represented exactly. Using base ten, any rational number which cannot be written with a denominator that is an exact power of 10 recurs, for example 1/3 is 0.333..., while 1/2 = 5/10 = 0.5. Similarly using base two, any rational

number whose denominator is not an exact power of two will require a recurring representation. If a floating point representation with a fixed precision is used, such numbers will be rounded. In BCD, as many decimal places as required can be used. Other benefits include quick conversion of BCD numbers to ASCII for display since each digit is in a byte. More circuitry is required for calculations in BCD, though the calculations are still straightforward.

Consider adding 9 and 8 in binary coded decimal. Adding the digits as one would usually gives

```
     1001
   + 1000
0001 0001
```

The first `0001` is a carry digit. If the number were in binary this would be sixteen, but since it only represents ten the extra six, `0110`, must be added whenever we have a carry digit:

```
     1001
   + 1000
0001 0001
   + 0110
0001 0111
```

This represents a one followed by a seven, giving 17 as required. For any carry digits or invalid BCD numbers (`1010` to `1111`), six must be added to give a valid BCD.

Other less well-known representations of the rational numbers are possible. One I have come across recently is the p-adic numbers [Horspool78]. They can represent any rational number exactly. It uses a compact variable length format. Addition, subtraction and multiplication are familiar and straightforward. Division is slightly more difficult.

The numbers are represented as a series $\sum d_i b^i$, where d_i are the digits and b is the base. The p-adic numbers do not require a decimal point and many of them involve an infinite, but repeating, sum. The natural numbers are represented as they would usually be in the chosen base. For example, using base two, 9 is `1001`. If a prime number is chosen as the base each rational number can be represented uniquely. For negative numbers using base two, note that

```
...11111
+  00001
...00000
```

This means –1 can be represented by `...11111`. Though this sequence repeats indefinitely, it repeats and so can be represented finitely. Some irrationals and complex numbers can be represented. Consider rational numbers. For example 1/3 is ...0101011, since three of them sum to unity:

```
...0101011
 + 0101011
 + 0101011
...0000001
```

[Horspool78] gives further details on this interesting representation of real numbers.

Further issues

This article has looked at representations of numbers on computers, various ways to compare floating point numbers, touched on issues concerning calculations, including the order of operations and tricks to compensate for rounding errors. Many other issues have not been considered including platform specifics, such as such as how to control IEEE rounding modes, and how to find out if an exception event has signalled. Other more general points have not been raised, including troubles that occur when floating point numbers are streamed into a string.

What does this output?

```
#include <iomanip>
#include <iostream>
int main()
{
  std::cout << std::fixed << std::setprecision(2)
     << 5000.525 << "\n";
}
```

VS2005 rounds down to 5000.52, when you might expect to get 5000.53. What does your compiler do with this?

```
double x = 0.00000000000001234567890123456789;
```

Various myths concerning floating point calculations abound. Hopefully it is now clear why the following are myths:

- Since C's float (or double) type is mapped to IEEE single (or double) precision arithmetic, arithmetic operations have a uniquely defined meaning across platforms.
- Arithmetic operations are deterministic; that is, if I do $z=x+y$ in two places in the same program and my program never touches x and y in the meantime, then the results should be the same.
- A variant: 'If $x < 1$ tests true at one point, then $x < 1$ stays true later if I never modify x'.
- The same program, strictly compliant with the C standard with no 'un-defined behaviours', should yield identical results if compiled on the same IEEE-compliant platform by different compliant compilers. [Monniaux08]

I am still left with the feeling I have more to learn and consider regarding floating point numbers. I suspect there is still much I do not fully understand or appreciate, however that is enough for now. As Wittgenstein said, 'Whereof we cannot speak, we must remain silent'. ■

Acknowledgements

A big thanks to everyone who helped review this – you know who you are.

References

[ACCU09] *accu-general* discussion, March 2009

[C++03] *C++ standard 2003*. 18.2.1.2

[Cook] http://www.johndcook.com/standard_deviation.html

[Goldberg91] David Goldberg 'What every computer scientist should know about floating-point arithmetic' *ACM Computing Surveys*, Vol 23(1), pp 5-48, March 1991

[Higham96] 'Accuracy and stability of numerical algorithms' Nicholas J. Higham, 1996

[Horspool78] R.N.S.Horspool, E.C.R.Hehner: 'Exact Arithmetic Using a Variable-Length P-adic Representation', *Fourth IEEE Symposium on Computer Arithmetic*, p.10-14, Oct 1978

[IEEE87] *IEEE Standard 754-1985 for Binary Floating-point Arithmetic*, IEEE 1987

[IEEE98] IEEE 754: 'An Interview with William Kahan' Charles Severance, *IEEE Computer*, vol. 31(3), p. 114-115, March 1998

[Knuth] Knuth. *The Art of Computer Programming*.

[Monniaux08] 'The pitfalls of verifying floating-point computations' David Monniaux May 2008. hal-00128124 http://hal.archives-ouvertes.fr/docs/00/28/14/29/PDF/floating-pointarticle.pdf

[Welford62] B. P. Welford 'Note on a method for calculating corrected sums of squares and products'. *Technometrics* Vol 4(3), pp 419-420, 1962 (cited in [Goldberg91])

[Wikipedia] http://en.wikipedia.org/wiki/Binary-coded_decimal

On Management: The Business Analyst's Role

Some management titles are poorly defined. Allan Kelly disentangles a knotty one.

In my previous article I looked at the role of Product Manager [Kelly09]. Usually this role only exists inside software product companies, that is, companies which produce software products which customers buy. But, as we saw in a previous article there are two other types of software development organization that need to be considered.

Inside corporate IT departments – which create software for the corporation's own use – and external service providers (ESPs, companies which write bespoke software for a specific customers) someone must still decide what is required. Even if requirements documents are not written there needs to be someone who decides what should be included in the software, what should not be included and what priorities should be assigned.

In such organizations this is the role of the Business Analyst, or BA. While the BA role has similar responsibilities towards the development team as a Product Manager the role is very different and requires different skills and experience.

Scrum – and other Agile methods – have a Product Owner role. This title describes a role in the method. In general Product Owner can be considered an alias for either a Product Manager or Business Analyst.

Difference between Product Managers and Business Analysts

The main difference between Product Managers and BAs is that the latter is inward facing. They look inside the company, they look at processes and practices inside a single company and how these can be improved through the use of software technology. In contrast Product Managers are outward facing, they look at the market and at multiple independent customers.

The second difference between the two roles is that BAs perform work in support of another. Somewhere in the organization there is someone – or some group – who wants different systems. Those systems will enable to the organization to reduce costs, improve service, or bring a new product to market. However, BAs are a supporting role, they are proxy customers for those who want to see the change happen.

Product Managers are usually responsible to someone else in the organization but they play the role of informed customer. For BAs the ultimate customer is the person whose budget pays for the IT work and/or receives the benefit of the work. If they had the time and skills they could take on the role themselves – after all, they are the ultimate owner.

In a small organization such a person may be able to play the Product Owner/BA role but as the organization grows they will find demands on their time prevent them from filling the role adequately. This is the time when a proxy customer should be introduced: someone who has the time and skills to fill the role properly.

Lesson 1: Product Owners need to give time to the role. Development teams need time from the Product Owner, and the Product Owner needs time to understand the issues.

Multiple hats

While the Product Manager role is often overlooked, the BA role is often misunderstood. This is because under the title of Business Analyst there are many different types of BA. Some of these roles carry the BA title erroneously; things would be simplified if some BAs were given the title 'System Analyst' or 'Assistant Product Manager'.

BAs may be called upon to fill a number of difference jobs all under the one job title. So an individual BA may on one project work as an internal consultant to help invent a solution. The next project may be somewhat simpler and only need the BA to capture requirements. Later in the same project the same BA would be well placed to help with the testing of the final software.

There is debate within the BA community itself as to the proper role for analysts. Therefore it is important not to look on the role either too narrowly or to assume that all BAs work the same way.

One definition of core BA role states:

> An internal consultancy role that has the responsibility for investigating business systems, identifying options for improving business systems and bridging the needs of the business with the use of IT. [Paul06]

In order to better understand the BA role it helps to first understand the different ways in which the title is used (and abused).

Non-IT Business Analyst

These Analysts have nothing to do with IT. They analyse businesses, business trends, markets, competitors and anything else that is connected with commerce and helps their employer. But they don't do it in order to create or change IT systems.

There is nothing wrong with this type of BA, it is just necessary to point out that not everyone with the title Business Analyst is concerned with IT systems.

Business Analyst as Systems Analyst or Software Designer

Before Business Analysts became a common in IT departments it was normal to find Systems Analysts. These analysts perform a similar function - deciding what should be in the system and what should not - but they focused on the computer system not the business. (Many Business Analyst are actually titled as Business Systems Analyst but for brevity the middle word gets dropped.)

This type of analysis creates a technology focus for work which should be business focused. Delivering a faster, better computer system should not

Allan Kelly realised after years at the code-face that most of the problems faced by software developers are not in the code but in the management of projects and products. He now works as a consultant and trainer, helping teams adopt Agile methods and improve development practices and processes. He can be contacted at allan@allankelly.net and http://allankelly.blogspot.net.

be the objective. Satisfying a business need – which might mean a faster better computer system – should be.

Lesson 2: Business Analysis is not System Analysis. If a company really needs a System Analyst then they should appoint one. (It should first explain to itself why the system analysis cannot be undertaken by the development team or and Architect.)

Some BAs, and Systems Analysts, are expected to take a role in designing the software. This is a misuse of the BA role. The BA role needs to focus on the business; software design should be left to the specialists.

Usually BAs do not have the skills to design software. In asking them to do so 'the business' may believe it is more likely to get what it wants from development teams. The net effect is to move some of the design activity from those who are best placed to do it to those who are less able.

Lesson 3: BAs should not design software. BAs should confine their role to analysis and determining what the business needs from the software and systems.

A BA who undertakes specification from a systems point of view, or undertakes system design is neglecting their true responsibility. When systems analysis is needed it is either the team as a whole or of a designated Architect who should undertake the work. Similarly software design is not the responsibility of the BA, it is either a responsibility on the whole team or on a designated Architect.

By concentrating on the business need the BA provides the team with room to create a solution. The more a BA specifies system constraints, or design details, the greater the restriction on the team. When this happens requirements contain design constraints.

Business Analyst as Product Manager

In some companies Product Managers exist, they fulfil the type of external analysis described in the previous article but they bear the title Business Analyst. This may be because the company has its origins in corporate IT – perhaps it was spun out of a corporate IT department.

Lesson 4: Some BAs are Product Managers with incorrect titles.

Occasionally companies that have both Product Managers and Business Analysts. Usually the BA is not really filling a BA role, they are a Junior Product Manager or they are a System Analyst. But occasionally it makes sense.

As we noted before there is a lot of work for a Product Manager to do. One way of reducing the work load is to appoint specialists to help them in some way. In these cases a BA works as an assistant to the Product Manager on some specific aspect of the product.

For example, a BA may pick up the competitor monitoring work from the Product Manager. The BA will able to spend more time monitoring, investigating and analysing the competition then feed this information to Product Manager.

Lesson 5: Some BAs are Assistants to Product Managers.

Business Analyst as Subject Matter Expert

It is possible to work in IT and yet know little about the application domain under development. For example, a Java developer may use their knowledge of Java programming for a bank, or an oil company, or for a telecoms company. A Project Manager may manage projects at an oil company one month and at an airline the next. While it helps with have domain knowledge it is often not essential. Skills like Java programming and project management are transferable.

Complex domains need individuals who have in depth understanding. Banks need banking experts; telecoms companies need communication experts, etc. etc. For these individuals skills such as coding are secondary to their knowledge of the domain. Such people are often known as Subject Matter Experts.

Lesson 6: Some BAs are Subject Matter Experts, their knowledge is more important in understanding what needs to be done than actual analysis.

Since the Business Analyst role entails understanding and describing organizations and technical domains Subject Matter Experts (SMEs) may gravitate to this title.

Yet it is not a foregone conclusion that a BA needs to be an expert in the domain to perform their role. Indeed, being an expert in current practices may blind one to opportunities for change. Starting an analysis assignment with an open mind, or blank sheet of paper, may be advantageous.

Lesson 7: The core BA skill is analysis: the ability to analyse, to understand a domain and a problem, then to communicate what needs to be done to change it. Therefore, it is not essential for a BA to have extensive domain experience.

Lesson 8: Some BAs are Subject Matter Experts and can fill their role because of deep knowledge. Other BAs are experts in analysis and fresh thinking; they can fill their role because of clear thinking.

The Business Analyst role

In the IT context a BA is someone concerned with information systems for business applications. The role is IT centric and extends beyond the pure technology system to include the processes and practices of people who use the system. As such it may also cover change management.

BAs are concerned with the analysis of systems, processes, practices and operations within an organization. They are tasked with understanding these things and proposing changes. But they are not tasked with designing those changes in detail.

In understanding these systems, and suggesting changes the BA is expected to take into account the overall business objectives and direction. It is unlikely the BA will have responsibility for revenue or cost saving but they will need to include these in their analysis.

As with the Product Manager the BA role entails requirements discovery. That is, enquiring and analysing the application domain to uncover the requirements of a technology system to improve the domain. Although it is likely the Product Manager will be far closer financial targets. Senior Product Managers may even have responsibility for meeting financial or other targets.

Sometimes the BA role is one of collecting business needs in order to communicate the needs to those responsible for technology solutions. On other occasions the BA acts as an internal consultant, helping the business understand what is possible and envisaging a different way of working by using technology.

Lesson 9: When a business knows what it wants the BA will collect the needs and communicate them to the technology providers. When a business is less clear on what is needed the BA will work as a consultant to analyse the problem and envisage a solution.

Given the variety of work expected of BAs finding a definition of the role is problematic. The authors who supplied the definition of BA work above [Paul06] noted this variety too:

> Although there are different role definitions, depending on the organization, there does seem to be an area of common ground where most business analysts work. ...to investigate business systems ...
>
> - To identify actions required to improve the operation of the business
> - To document the business requirements for the IT system"

(Paul and Yates 2006)

Product Owner role and the team

As already noted, in the Agile teams - and particularly for teams following Scrum - it has become common to talk of a Product Owner. Yet outside of the description of the Agile process these methods say little about what the Product Owner does. Or rather, vanilla-Scrum and vanilla-XP only describe what the role does in the process.

Outside of the time the Product Owner spends in the Scrum process they need to use their time to support the process so they can speak from a position of knowledge. To do this the Product Owner needs the skills of a Product Manager or Business Analyst.

For the development team it matters little whether the Product Owner is a BA or Product Manager, the role is the same. The Product Owner decides what goes into a product, in what order and approves what is created. (To put it in technical terms, both BAs and Product Managers are implementations of the Product Owner role.) However there is a big difference in how BAs and Product Managers fill this role.

Lesson 10: A Scrum (and other Agile methods) Product Owner should normally be a Product Manager or a Business Analyst. In general, the title Product Owner can be considered an alias for Product Manager or Business Analyst.

On rare occasions the Product Owner may be from another background. They might, XP-style, be an actual customer. Or, for product which is technical facing the Product Owner may come from a technical background. For example, a Scrum team developing display drivers may have a video engineer as a Product Owner. However before accepting a technician as Product Owner look for alternatives first.

Just to complicate matters, and having spent this and the previous article sketching the two different roles the world is changing. Once up on a time the difference between a product company employing Product Managers and a corporate IT department employing BAs was clear. Now however the two are merging.

Consider for example an imaginary package travel company. Prior to the year 2000 all package holidays were sold via travel agents or over the telephone. IT systems existed to track customers, manage inventory, handle administration and so on. These were for internal use only so it made sense to use BAs for analysis.

Starting in 2000 the company has increasingly sold holidays over the web. The number of agent bookings has declined, retail stores have been closed but online bookings have soared. The web systems which handle these bookings are extensions to existing internal systems. To the end customer this internal/external difference is meaningless. Part of the holiday experience is the booking process.

In such a situation it is not clear whether the company should use BAs or Product Managers. To use BAs may neglect the customer experience side, while to use Product Managers may neglect the internal aspects. A mix of skill sets is required.

Lesson 11: Product Owners increasingly need a mix of Business Analyst and Product Manager skills.

Product Manager, Project Manager or Business Analyst?

As if the confusion between BA, Product Manager and Product Owner were not enough there is a third role which is sometimes added to the mix: the Project Manager.

Like Business Analyst the Project Manager role is somewhat ill-defined and quite elastic. The standard text for the PRINCE 2 method (a popular project management technique in the UK) does not define the role of Project Manager [TSO05].

As a rule-of-thumb BAs and Product Managers are concerned with the ‘what’ of a development while Project Managers are concerned with the ‘when’. Beyond this different organizations slice-and-dice responsibilities differently.

While PRINCE 2 contains processes to ensure the ‘what’ and ‘why’ are defined (in the business case) it is silent on who should create these documents. Neither is there any guidance on how to create a business case or what skills are required to write one. This is because writing a business case is not a project manager's responsibility.

Lesson 12: Project Managers are not Product Owners, neither are they Business Analysts or Product Managers. Project Managers have different training, different objectives and often different experiences.

To complicate matters further, some companies use the title Architect for Product Managers. At the end of the day it is better to look at what individuals actually do rather than their title when trying to understand roles.

Finally

Hopefully this article and the previous one have gone some way to clarifying and disentangling the roles of Product Manager and Business Analyst. Along the way I hope to have convinced you that the Product Owner role is important – whether it is filled by a Product Manager or BA – and that there is more to it than meet the eye. Consequently it should be clear that these are not roles developers can do in their spare time. ■

References

[Kelly09] Kelly, A. 2009. ‘The Product Manager role’ ACCU *Overload* (90).

[Paul06] Paul, Debra and Yates, Donald. 2006. *Business Analysis*. Swindon: British Computer Society.

[TSO05] Commerce, Office of Government. 2005. Managing Successful Projects with PRINCE2. Fourth Edition. London: TSO (The Stationary Office).

Complexity, Requirements and Models

Programs can be unnecessarily complex. Rafael Jay examines a technique for doing better.

You have the requirements for a new feature. Your customers need it done on time. You start designing. You know it will touch a few areas of your codebase so you start by looking at those. It's hard to get a clear picture of how they work so you draw a few class and sequence diagrams. Eventually you think you understand, but there's a lot of detail and you have to keep referring back to your diagrams. Deadlines are looming so you start coding. A week in you realise that strange dependency which didn't seem important actually disguised a key connection to another part of the system. Your design doesn't quite work. You don't have time to go back to the drawing board so you hack something together. You feel bad but you have to meet your customer commitments.

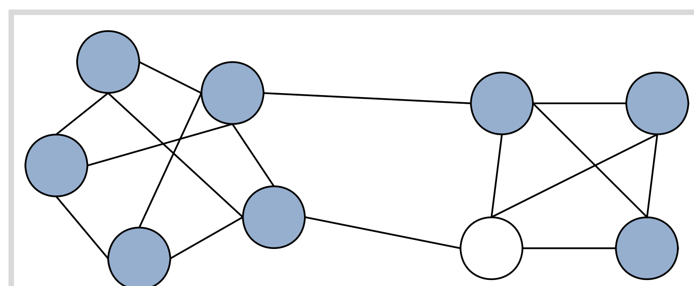
Next time round you're wiser. You know that code is tricky so you ask your manager for a couple of extra weeks to refactor it first. But refactor it into what? It's hard to draw a boundary around the area you're looking at and other parts of the code. And it's not entirely clear what you should refactor it into anyway – where should each class go? You do your best to apply some patterns and good software engineering techniques, but you keep having to back off because of unexpected dependencies. Two weeks later you've cleaned up a few localized issues, but you're not convinced you made that much difference overall. And behind that you have a nagging suspicion that this wasn't even the worst part of the codebase. Maybe you should have spent the time refactoring something else.

If these experiences sound familiar, your code is probably too complex. But what is complexity? We know what too much feels like: you discover one more thing to think about and suddenly your head explodes. You can't keep a clear picture of how it all fits together any more. This kind of anecdotal measure – how much it makes your head hurt – is perfectly valid and there's no reason why you shouldn't use it as an input into your refactoring efforts. But it's hard to compare different developers' sore heads to identify the most problematic areas, and it doesn't offer much insight into why their heads are hurting. We could do with something a bit more scientific.

Kettles

One explanation of complexity I have often found useful comes from Christopher Alexander's 1964 book *Notes on the Synthesis of Form* [Notes]. This book is concerned with the design of complex artefacts, which I'm sure most people would agree includes software. Alexander invites us to consider each requirement of an artefact as a light bulb which goes off only when the currently proposed design satisfies that requirement. For example, a kettle must be of adequate capacity, durable, not too heavy to lift, cheap to make, and many things besides. The initial design – a blank sheet of paper – actually satisfies some of these. A non-existent kettle never wears out, is easy to lift and cheap to make. So the light bulbs representing those requirements are off. However it does not have adequate capacity, so at least one light bulb is on. The design still needs some work.

Rafael Jay has been programming professionally for over a decade. He is a technical lead at Trayport Ltd, a software house developing trading platforms in C++ and .NET. He can be contacted at rafaeljay@bcs.org.uk



Light bulbs and connections. Each bulb represents a requirement. When it is off, the requirement is satisfied. While a bulb is on, there is a possibility that each connected bulb may also come on. This represents the possibility that addressing one requirement may inadvertently break others.

Figure 1

Requirements are interconnected. Let's say we redesign the kettle so it's made of finest titanium with a capacity of two litres. The capacity light bulb goes out, but titanium is expensive, so the cost light bulb comes on. Alexander models connections between requirements by connecting their bulbs. Each connection between two bulbs implies a certain probability that while one of the bulbs is lit, the other bulb will also light up. This represents the probability of breaking the connected requirement while trying to fix the original. (See Figure 1.)

A software product, like a kettle, is an artefact that must meet certain requirements. This is why we typically start off with a functional requirements document, or a set of stories and acceptance tests, or at any rate some kind of breakdown of what the product is actually meant to do. We can model those requirements as light bulbs and connections.

Let's look more closely at the connections between requirements. If not for connections, it wouldn't matter how many requirements a product had, because we could address them all as independent, trivial problems. This might take a long time if there were a large number, but the challenge would be perseverance rather than brainpower. At the other end of the scale, a product where every requirement was connected to every other – where work to address one requirement could potentially break all the others – would rapidly exceed the capacity of our limited human minds.

Light bulbs

Alexander illustrates these issues by considering various systems of interconnected light bulbs where, in any given second, there's a 50% probability of any lit bulb turning off and a 50% probability of a bulb coming on if its neighbour is on. For each system he asks how long, given an initial stimulus of a single bulb coming on, it will take for the ripples of illumination to die out – how long the system will take to reach an equilibrium state where every bulb is off. In software terms this is

A connection between lights indicates a possibility that work done to address one might break the other

equivalent to asking how long after fixing a bug you'll be able to declare the work, and all its ramifications, finished. Unsurprisingly, the more numerous and stronger the connections, the longer it takes to reach equilibrium. In a system of one hundred maximally interconnected bulbs, Alexander calculates it will take longer than the lifetime of the universe for them all to go off. This equates to a never-ending cycle of design, realise you've broken some requirements, redesign, realise you've broken some more requirements, and so on. The lesson we can draw is that there's no point trying to address design problems with large numbers of densely interconnected requirements – we simply don't have the intellectual capacity to solve them.

Nevertheless, a glance at any sizeable requirements document will confirm that we frequently do address software design problems that have hundreds of requirements. The key, according to Alexander, is how effectively the system decomposes into independent groups. He calculates that a system of one hundred light bulbs decomposed into ten distinct groups – densely and strongly interconnected within each group, but not between groups – would take about fifteen minutes to reach equilibrium. Fortunately for us as developers, and indeed as humans, most of the design problems we face fall more into this latter category than the former. Their requirements decompose naturally into smaller subgroups which we can solve as more or less independent problems.

Software: requirements

Let's look at an example. An online retailer handles orders, which link stock items to customers who want to buy them. It's not hard to see how the functional requirements for this part of the product would decompose into groups for orders, items and customers (Figure 2).

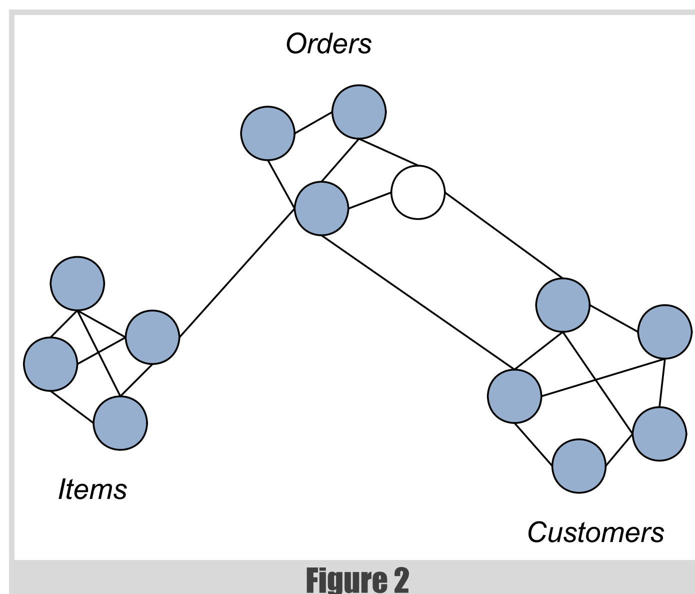


Figure 2

The groups are not arbitrary. Each exists because the requirements cluster around an abstract domain concept, such as orders, items or customers. The domain concept explains why the requirements in each group are tightly linked, why work done to address one of them is likely to affect the others. Thus in many ways an Alexander-style requirements diagram like the one in Figure 2 is equivalent to a domain model, or models, from the world of domain driven design.

Alexander's approach, however, can give us more of an insight into the artefact's complexity. The number of requirements and the number, strength and pattern of connections between them gives us the basis of a less subjective measure of complexity, one we can use to compare different products and to compare different parts of the same product. We can expect it to have a real correlation with how much working on those areas will make our heads hurt.

Sadly it isn't adequate. I have worked on a number of products with broadly comparable complexity of requirements, and some of them have been a lot more painful than others. The number of requirements and connections between them tell us how complicated a product has to be – the necessary degree of complexity inherent in an artefact capable of satisfying all those requirements simultaneously. But by itself it does not completely measure the thing we wanted to measure: the size of the headache it will give us as developers.

The humble developer

Thus far we have focused almost exclusively on the artefact – the software. But what about the developer? Complexity is only a problem because the human mind has limits. We can only hold so many things in our heads at once. To understand the problem of complexity we must examine not only the artefacts but also the people who design them. What do developers do?

The bulk of our time as developers is spent implementing new features or fixing bugs. A bug means a requirement of our software product isn't satisfactorily met – a light is on. A feature means one or more new lights are 'plugged in' to the existing ones, with at least one of the new lights being on (otherwise there's no new work to do – the feature is already provided by the existing product). Our job as developers is to make all the lights go off again, to fix the bug or implement the new feature without breaking the implementation of any of the other features.

To do this we need to know not only which lights are currently on, but also which lights those ones are connected to. A connection between lights indicates a possibility that work done to address one might break the other. As developers we need to be aware of that possibility so we can check the corresponding areas of code and adjust them as necessary. For example, if I change how we store a customer's date of birth to accommodate a new feature, I might need to change connected parts of the existing code to compensate. If I miss a connection, I risk introducing a regression bug – a requirement which used to be met but which I've now inadvertently broken. We're all familiar with the cost of these, particularly if they go unnoticed until later stages in the production process. However there's also a cost to false positives. If I believe there's a connection when there isn't, I waste time investigating irrelevant code. This is harder to measure than

> The code is the only thing that . . . you can guarantee a developer will have to look at to accomplish their work

regression bugs, but it nevertheless siphons off development time that could more usefully be spent implementing new, saleable features.

Of course it's not just the immediate connections that I need to worry about. If work on my immediate target causes a directly connected light to come on, then there's a chance that work done to turn that light off will trigger further lights to come on; and work on those further lights may in turn trigger others. I have to chase an expanding wave front of broken requirements through the code base. It's at this point that knowledge of the groups of requirements is very useful.

The groups can help

Groups of requirements exist whether we're aware of them or not. The online retailer system has groups of densely interconnected requirements for items, customers and orders regardless of whether I, as a developer, know that those groups exist or take them into account in my design and implementation. The fact that they do exist means that even if I'm not aware of them I'll probably reach a satisfactory design for the system eventually, turning all the lights off, because the wave fronts of broken requirements will be naturally limited. But the process is likely to be clumsy and time-consuming, tackling each broken requirement in a random order, constantly having to remember all the details of which lights are connected to which; and most likely getting it wrong fairly frequently due to the number of things I have to bear in mind all at once. The end result is usually features that throw up a barrage of unexpected extra work items as you implement them, then haunt you with regression bugs for months or even years afterwards.

If, on the other hand, you know which groups the immediate targets are in and how those groups relate to other groups then you can design a lot more rationally. You know that you need to consider the other requirements in the target groups, and pay especial attention to the points where those target groups connect to other groups – the interfaces. For example, on changing how we represent a customer's date of birth, I know I need to look closely at the code which implements the other customer requirements, and keep a tight grip on the interfaces between the customer code and other code which depends on it, preferably leaving the interfaces between them unchanged. I also know that I don't need to look at code which implements unrelated groups of requirements. This allows me to focus my limited development time much more closely on those areas where it's most needed.

Knowledge

So, to do a good job as a developer you need to know the connections between requirements and how they decompose into groups. Where does this knowledge come from?

There are three main repositories: external documentation such as design documents, requirement specs, etc; other developers; and the code itself. It's here that the ways in which software is not like a kettle start to become important. A kettle is typically designed once, probably by a single individual, based on requirements that change very little over time. Most substantial software applications, by contrast, are continually redesigned by an ever-changing team of developers to meet ever-expanding requirements. Under such circumstances you cannot rely on external documentation to stay up-to-date, or the other developers to have a full and accurate knowledge of how the system works. The code is the only thing that absolutely has to be complete and accurate, and moreover it's the only thing that you can guarantee a developer will have to look at to accomplish their work. This is not to say that documentation and other developers don't have an important part to play, but the code is the most effective repository of knowledge about requirements and how they decompose into groups. The easier you make it for developers to acquire this knowledge from the code, the better and faster they will be able to work.

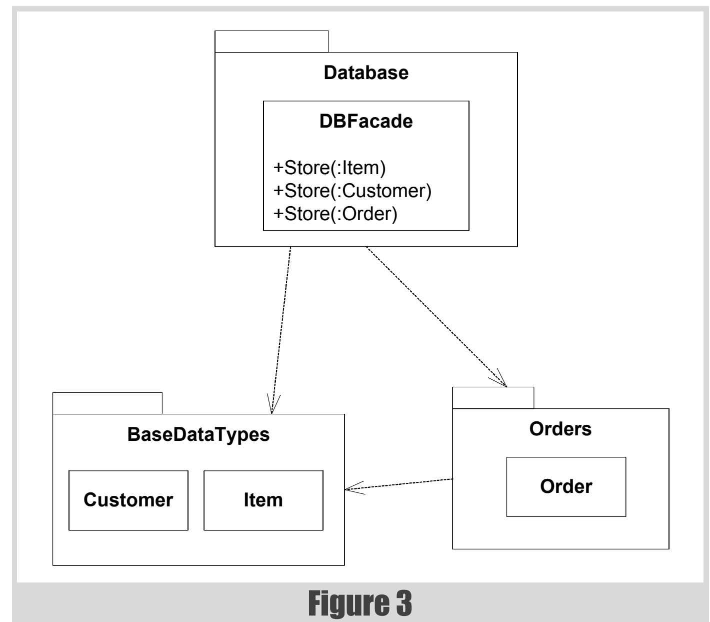


Figure 3

Software: design

Let's look at a first draft design for the online retailer's system (Figure 3).

The `Customer` and `Item` classes go together in a `BaseDataTypes` module because they're the basic system types, combining to make up an `Order`. I've also added a `Database` module because there are requirements that `Customers`, `Items` and `Orders` be permanently recorded.

There are two main things wrong with this design, things that might well flummox a developer coming fresh to the code. Firstly, the `Database` module implements each data type's persistence requirements. Effectively it implements a light bulb from each of the customer, item and order requirements groups. This is confusing because we now have to look in two places to identify all the requirements in each data type's group. Furthermore, it's not obvious that we need to look in two places. As a developer I expect a `Database` module to be about databases – integers, strings, tables, columns and whatnot. I don't expect it to implement

requirements for `Customers`, `Items` and `Orders`. There's a fair chance that if I change one of those data types I'll forget to check that its persistence code still works, introducing a regression bug.

The second problem is with the `BaseDataTypes` module. It crowds together the `Customer` and `Item` implementations, suggesting to the unwary developer that the customer and item requirements might be connected. In fact, as we saw above, they are not. But because the code is together it may well take some valuable time to reach this conclusion. Moreover, a developer tasked with changing customer or item requirements will have to spend time hunting for the relevant code. 'BaseDataTypes' doesn't give you much of a hint as to what the module contains. Nor will it help much when deciding where new code should go. What exactly is a 'base' data type? Modules with this kind of ill-defined name often end up as dumping grounds for all manner of more or less unrelated code.

The above design makes it hard to find the groups of requirements by looking at the code. Even when you've found them, you have to constantly remember that `Database` implements some persistence requirements, and `BaseDataTypes` holds two separate groups. This uses up valuable brain capacity before you even start work. Of course in a small example like this it's not too much of a problem. But in large software products the burden of translating between the code on the screen and the working model in your head can become an enormous drag on development activity. In severe cases it simply takes too long to deduce a useful working model from the code and you end up hacking away blind, hoping the compiler will tell you if you do something wrong.

Software: design revisited

Let's have another go (Figure 4).

The `Database` module now represents a database and nothing else. Each data type is responsible for storing itself in the database. The requirement groups for customers, items and order are now each implemented in a single module, rather than split across two, and it's easy to find where each is implemented because the modules are appropriately named.

There are still issues with this design. In particular it's questionable whether the data types should need direct knowledge of the database. Such inappropriate dependencies are often a good warning sign that there's further work to be done, and it's much easier to spot them if the modules themselves make sense. It's much easier to question why `Customers` depends on `Database` than why `BaseDataTypes` depends on `Database`, because it's much clearer what `Customers` is all about. Nevertheless, from a complexity point of view, this new design is a big step forward from what we had before.

Complexity and modules

We saw above that the number of requirements and the number, strength and pattern of connections between them serves as a useful measure of how complex a software product has to be, but is inadequate as a measure of how complex it actually is. It doesn't fully explain why some products make developers' heads hurt so much more than others. The missing factor seems to be ease of perception: how hard it is for a developer to perceive the requirements and their connections by looking at the code. The more brainpower you have to spend constructing and holding onto an accurate mental map of what requirements are implemented where, the less brainpower is left over to actually reason about them. You are more likely to make poor design decisions, increasing the complexity and making the next feature even harder to implement.

The obvious conclusion is that we should try to keep actual complexity as close as possible to necessary complexity – make it as easy as possible to see the requirements through the code. And it's particularly important to help developers identify the groups into which those requirements decompose. As our online retailer example demonstrated, modules have a vital role to play here.

A good module is one which implements all the requirements from a particular requirements group and no others, and is named after the abstract domain concept behind that group. Modules with these qualities make it very easy for a developer to see the requirements and their connections through the code. In our initial design above, none of the modules were good. The `Database` module implemented all the database requirements, but it also implemented requirements from the data types' groups. Correspondingly, `Orders` and `BaseDataTypes` were missing some of the requirements that rightly belonged to them. `BaseDataTypes` was trying to hold two unrelated groups; unsurprisingly therefore it wasn't named after any kind of recognizable domain concept. A good rule of thumb for module quality is to ask how easily, given its name, you could decide whether the module would be involved in any particular feature or bug. For good modules it should be easy.

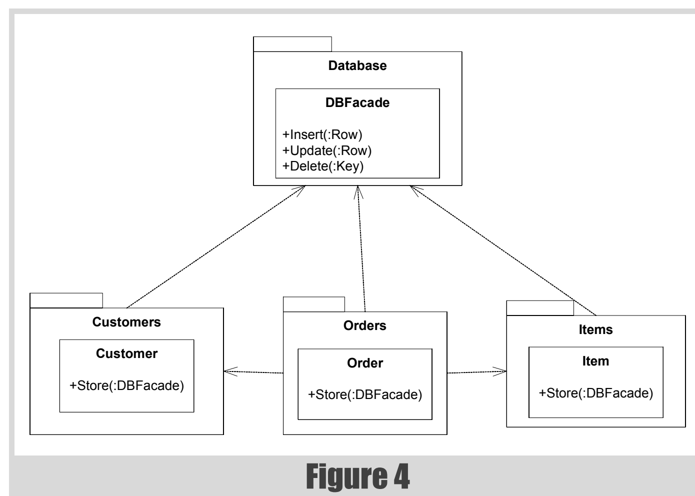


Figure 4

In conclusion

Let's revisit the opening scenarios for a code base with good modules.

You have the requirements for a new feature. Your customers need it done on time. You start designing. The domain concepts used to describe the feature tie in naturally with the domain concepts used to name the modules, so it's easy to see which parts you need to start looking at and how they relate to the rest of the code base. Because each module represents a single, coherent domain concept it's proved easy for your colleagues and predecessors to add a little high-level documentation to each, so you quickly get to grips with any parts you don't already know. The time you save locating and understanding the code can now be spent designing your new feature; and you can do a better job because you understand that code more clearly. Ultimately you implement your feature faster and leave the code in a better state.

Of course there are still problem areas you want to refactor. Armed with a deeper understanding of complexity you can survey the code base for the worst areas. Which modules have incoherent names such as 'BaseDataTypes'? Where is the coupling between modules unexpectedly high, perhaps indicating that some bits of code are in the wrong place? Are there dependencies which don't sound right, such as `Database` depending on `Orders`, or vice versa? When you have a vision of what the code base should look like it becomes easier to identify and prioritize the problems and come up with appropriate solutions. Furthermore if your whole team shares the vision then all your refactoring efforts fit in with each other. A virtuous circle of reduced complexity and better design begins. ■

References

[Notes] *Notes on the Synthesis of Form*, Christopher Alexander, Harvard University Press 1974. ISBN 0674627512.

An Introduction to FastFormat (Part 3): Solving Real Problems, Quickly

A good library must be useful in practice. Matthew Wilson looks at usability and extendability.

This article, the third and last of the current series on the FastFormat formatting library, discusses several use cases, from real world projects and discussion forums, that illustrate how the library can be used to achieve concise, transparent application code while utilising its flexibility and performance advantages. It is about solving real formatting problems, quickly: both in speed of development and speed of executed code.

Along the way, we'll look at some of the more esoteric aspects of the application layer, customisation of the format specification defect handling, and consider cases where suppressing unused argument exceptions is useful.

Introduction

This article is divided into two halves. The first half describes six use cases, four of which are from real applications, four of which demonstrate improvements to application code, and four of which involve performance benefits. (Not the same four.) I'll present performance measures for each scenario for which it is relevant, putting more flesh on the performance characteristics suggested in the Yaffle scenario from part 1 [FF1]. The second half of the article is a mix of miscellaneous but useful tips for taking your use of the library further. Along the way we'll see examples of FastFormat interoperability with MFC, ATL, and the Pantheios logging library [PAN].

The first two scenarios are pedagogical, contrasting the use of FastFormat for formatting columnar floating-point data, and in formatting according to absolute tabulations. The benefit of using FastFormat in these cases is primarily in performance.

The next two are extracts from client codebases, and involve server connection logging and database insert statement preparation. These demonstrate improvements both in application code transparency and performance.

The last two scenarios do not involve performance improvements. (Well, they might, but that's unimportant.) Rather, they involve substantial improvements in the transparency of application code by dint of FastFormat's expressiveness and flexibility.

FastFormat in action

Several of the scenarios described here are extracted from clients' work, and so names have been changed and types simplified. Also, the samples assume the inclusion of the fastformat/ff.hpp header to alias the **fastformat** namespace to **ff**, and the inclusion of whatever other headers are required for the various types and libraries involved. The full source of all the programs are included with the FastFormat distribution (version 0.4+).

Except where stated otherwise, the performance of each scenario was ascertained by invoking each of the statements 10,000 times, and repeating that loop three times, taking the times on the third outer iteration to minimise environment effects (since we're writing to **stdout**). The output was piped to a bit-bucket program – just a **getchar()** loop – to remove the latency of writing to the console/terminal from the measured times. The results are presented in milliseconds. The tests were conducted on Mac OS-X with GCC 4.0 (32-bit), on Linux with GCC 4.1 (64-bit), and on Windows with Visual C++ 9 (32-bit).

Floating-point columns

The first scenario is based on a question about C++ formatting on StackOverflow [SO], which asked how to do the following using the IOStreams:

```
printf("%-14.3f%-14.3f\n", 12345.12345,
12345.12345);
```

resulting in the output (where · represents a space):

```
12345.123·····12345.123·····
```

The implementations for IOStreams, Boost.Format, and FastFormat are shown in Listing 1. Note that I've added enclosing square braces as an aid to verifying that they all produce identical output. (This was done simply by running them all through **uniq**, a little trick I learned from a UNIX guru long ago.)

All present clear and transparent code except, in my opinion, the IOStreams, due to the verbosity of the code and the inconsistent semantics between the manipulators **setprecision** (sticky) and **setw** (non-sticky). Of course, some may argue that the explicit nature of something like **setprecision** is far more transparent than an arcane squiggle such as **"%-14.3f"**. So the issue of transparency here is somewhat subjective.

Totally objective, however, are the performance results, in milliseconds, are shown in Table 1. It's no big surprise to see that Streams is the standout performer here (since all the others form their floating-point arguments in terms of **sprintf()**).

Times (in ms) for 10,000 invocations of the Floating-Point Columns scenario

Library	GCC 4.0 (32)	GCC 4.1 (64)	VC++ 9 (32)
Streams	14.7	33.5	43.9
IOStreams	81.1	91.8	152.2
Boost.Format	103.3	104.0	223.8
FastFormat.Format	21.6	42.5	64.7
Boost.Format (1-arg)	109.7	105.6	224.9
FastFormat.Format (1-arg)	13.4	27.2	38.9

Table 1

Matthew Wilson is a development consultant specialising in performance and robustness, and author of numerous articles, books, and open-source software libraries. He prides himself on writing faster software than anyone else, yet is abashed that his books are slower to write (and to sell) than everyone else's. He can be contacted at stlsoft@gmail.com.

The issue of transparency here is somewhat subjective. Totally objective, however, are the performance results

```
// Streams
printf(
  "[%-14.3f%-14.3f]\n"
, 12345.12345
, 12345.12345
);

// IOStreams
std::cout
  << '['
  << std::setiosflags(std::ios::fixed)
  << std::left
  << std::setprecision(3)
  << std::setw(14)
  << 12345.12345
  << std::setw(14)
  << 12345.12345
  << ']'
  << std::endl;

// FastFormat.Format
ff::fmtln(
  std::cout
, "[{0}{1}]"
, ff::to_f(12345.12345, -14, 3)
, ff::to_f(12345.12345, -14, 3)
);

// Boost.Format
std::cout
  << boost::format("[%-14.3f%-14.3f]\n")
    % 12345.12345
    % 12345.12345;
```

Listing 1

Since **Boost.Format** and **FastFormat.Format** both allow an argument to be reused, we can simplify the statements and just have one argument, as in Listing 2 (floating-point column solutions reusing a single parameter). While there's little realism for the current scenario, there are cases where it is useful to use arguments multiple times, so it's interesting to see the effect. The times are shown in the bottom two rows of Table 1.

As expected, FastFormat's time drops, making it even faster than Streams for this edge case. The **Boost.Format** times stay almost exactly the same, so we may assume that it must perform the argument conversion twice.

Tabulations

One of **Boost.Format**'s advanced features is the ability to apply absolute tabulations, something none of the other examined libraries is able to do. The example given on the library's website, assumes three vectors of strings, such that when used with the following statement

```
// FastFormat.Format
ff::fmtln(
  std::cout
, "[{0}{0}]"
, ff::to_f(12345.12345, -14, 3)
);

// Boost.Format
std::cout
  << boost::format("[%1$-14.3f%1$-14.3f]\n")
    % 12345.12345;
```

Listing 2

```
  std::cout
    << boost::format("%1%, %2%, %|30t|%3%\n")
      % forenames[i]
      % surnames[i]
      % tels[i];
```

the output is as follows:

```
  Marc-François Michel, Durand, 0123 456 789
  Jean, de Lattre de Tassigny,  0987 654 321
```

As I mentioned in part 1 [FF1], FastFormat is able to support absolute tabulations with a little indirection. The above output can be obtained as shown in Listing 3 (synthesising absolute tabluations with **FastFormat.Format**)..

```
std::string scratch;

ff::fmtln(
  std::cout
, "{0,40,,<}{1}"
, ff::fmt(
    scratch
  , "{0}, {1}, "
  , forenames[i]
  , surnames[i]
  )
, tels[i]
);
```

Listing 3

Clearly it's not as transparent as the **Boost.Format** statement, but it's not opaque either. And given the relative performances (Table 2), the picture's not too bad.

Server connection Log

This next scenario is extracted from a client's proprietary internetworking server (UNIX and Windows), which writes out connection event logs containing connection identifier, addresses + port, time and bytes transferred. Consider the following fictionalised structure:

Providing equivalent functionality is going to involve extra effort

Times (in ms) for 10,000 invocations of the Tabulations scenario

Library	GCC 4.0 (32)	GCC 4.1 (64)	VC++ 9 (32)
Boost.Format	237.2	141.8	366.8
FastFormat.Format	37.0	46.0	149.1

Table 2

```
struct connection_t
{
  std::string     connectionId;
  struct in_addr  remoteAddress;
  struct in_addr  localAddress;
  unsigned short  port;
  unsigned long   numBytesTransferred;
  struct tm       completionTime;
} conn;

conn.remoteAddress.s_addr = htonl(0xC0A8A0f7);
conn.localAddress.s_addr  = htonl(0x7f000001);
conn.port                 = 5651;
conn.numBytesTransferred  = 102401;
conn.completionTime       = . . . // now
conn.connectionId         = "channel-1";
```

Currently, the format of the log is (though this may change):

```
<id> <time> <remote-addr> <local-addr> <port>
<bytes>
```

giving an output along the lines of:

```
channel-1 May 03 03:50:41 2009 192.168.160.247
127.0.0.1 5651 102401
```

This can be achieved simply with both FastFormat APIs, as shown in Listings 4 and 5. In this case I think the `FastFormat.Format` version is more transparent, and makes reordering of the replacement parameters a trivial matter (as suggested by the non-sequential format string in the example).

- Listing 4 shows a `FastFormat.Write` implementation of the Server Connection Log scenario
- Listing 5 shows a `FastFormat.Format` implementation of the Server Connection Log scenario

With Streams, IOStreams or any other library that does not support the automatic insertion of `struct in_addr` and `struct tm`, providing equivalent functionality is going to involve extra effort. Consider the Streams version (Listing 6), which is pretty close to the original implementation. There's considerably more code, and it's clear why the log format was originally difficult to change.

The brittleness of the Streams example format string can be obviated by using IOStreams, although it does require the definition of two inserters

```
void log_connection(connection_t const& conn)
{
  ff::writeln(
    stm
  , conn.connectionId
  , " "
  , conn.completionTime
  , " "
  , conn.remoteAddress
  , " "
  , conn.localAddress
  , " "
  , conn.port
  , " "
  , conn.numBytesTransferred
  );
}
```

Listing 4

```
void log_connection(connection_t const& conn)
{
  ff::fmtln(
    std::cout
  , "{0} {5} {1} {2} {3} {4}"
  , conn.connectionId
  , conn.remoteAddress
  , conn.localAddress
  , conn.port
  , conn.numBytesTransferred
  , conn.completionTime
  );
}
```

Listing 5

(Listing 7: IOStream Inserters for `struct tm` and `struct in_addr`) in order to cut down on the amount of application code.

With these, we can now write a much improved version using IOStreams (Listing 8).

It's worth noting that the compatibility for `struct tm` and `struct in_addr` that is afforded to FastFormat (and other libraries, such as Pantheios [PAN]) by STLSoft's string access shims (see [FF2, XSTLv1, IC++] for more details) is automatic. You don't have to define anything to make use of it, merely ensure the right `#include`s are made. This is a clear win for FastFormat over IOStreams in the case of `struct in_addr`, since the dotted-decimal format (e.g. 127.0.0.1) is widely accepted. However, with `struct tm`, it is only convenient by accident, since the string access shim format – a là `strftime()` – is equivalent to that required by the server log. If any other format was required, then you'd either have to write an inserter function or customise via the

However, it is only convenient by accident

```
void log_connection(connection_t const& conn)
{
  char    time[21];
  size_t  n0 = strftime(
      &time[0], STLSOFT_NUM_ELEMENTS(time)
    , "%b %d %H:%M:%S %Y"
    , &conn.completionTime);
  STLSOFT_ASSERT(
     n0 < STLSOFT_NUM_ELEMENTS(time));

  uint32_t ra_l =
     ntohl(conn.remoteAddress.s_addr);
  uint32_t la_l =
    ntohl(conn.localAddress.s_addr);

  fprintf(
    stdout
  , "%.*s %.*s %d.%d.%d.%d %d.%d.%d.%d %d %lu\n"
  , int(conn.connectionId.size())
      , conn.connectionId.data()
  , int(n0), time
  , ((ra_l >> 24) & 0xff),
    ((ra_l >> 16) & 0xff),
    ((ra_l >>  8) & 0xff),
    ((ra_l >>  0) & 0xff)
  , ((la_l >> 24) & 0xff),
    ((la_l >> 16) & 0xff),
    ((la_l >>  8) & 0xff),
    ((la_l >>  0) & 0xff)
  , conn.port
  , conn.numBytesTransferred
  );
}
```

Listing 6

filter_type mechanism (see [FF2]), each of which is equivalent to the effort of defining an inserter.

The relative performances are shown in Table 3. It's no surprise that the IOStreams solution fairs poorly, but it is interesting to see how FastFormat (in either guise) is on a par, or even better in some cases, than Streams in a non-trivial real-world example. Given the substantial differences in transparency of the code, FastFormat would appear to be a clear winner in this case.

Database insert statement

This next scenario is from a client's (UNIX) codebase. The particular code is to form a database insert statement, part of a high volume data processing subsystem. The client does, er, financial things, and they're extremely cagey about their work, so I hope you'll forgive the heavy obfuscation of names and types.

```
inline std::ostream& operator <<(
  std::ostream&     stm
, struct tm const&  t
)
{
  char    time[21];
  size_t  n0 = strftime(
      &time[0], STLSOFT_NUM_ELEMENTS(time)
    , "%b %d %H:%M:%S %Y"
    , &t);
  STLSOFT_ASSERT(n0 <
     STLSOFT_NUM_ELEMENTS(time));
  return stm.write(time, n0);
}

inline std::ostream& operator <<(
  std::ostream&         stm
, struct in_addr const& addr
)
{
  uint32_t ra = ntohl(addr.s_addr);

  return stm
      << ((ra >> 24) & 0xff)
      << '.'
      << ((ra >> 16) & 0xff)
      << '.'
      << ((ra >> 8) & 0xff)
      << '.'
      << ((ra >> 0) & 0xff);
}
```

Listing 7

```
void log_connection(connection_t const& conn)
{
  std::cout
    << conn.connectionId
    << ' '
    << conn.connectionTime
    << ' '
    << conn.remoteAddress
    << ' '
    << conn.localAddress
    << ' '
    << conn.port
    << ' '
    << conn.numBytesTransferred
    << std::endl;
}
```

Listing 8

The primary discriminant in this case is performance

Times (in ms) for 10,000 invocations of the Tabulations scenario

Library	GCC 4.0 (32)	GCC 4.1 (64)	VC++ 9 (32)
Streams	53.0	59.5	86.6
IOStreams	94.4	111.3	248.5
FastFormat.Format	56.0	55.0	86.7
FastFormat.Write	49.7	47.8	86.5

Table 3

Times (in µs) for 10,000 invocations of the Database Insert Statement scenario

Library	GCC 4.0 (32)	GCC 4.1 (64)	VC++ 9 (32)
IOStreams	1210	1546	5378
FastFormat.Format	533	456	569
FastFormat.Write	468	396	530

Table 4

The original implementation was done using **`std::stringstream`** to form the statement, along the lines shown in Listing 9.

The **`member_??`** variables are all integers. The **`makeSlice()`** helper function converts an integer to a string except where it is equal to the application-defined sentinel constant **`intNan`**, in which case it is converted to the string **`"0N"`**, as in:

```
const int   intNaN = 0x7fffffff;

class BusinessAdaptor
{
  . . .
public:
  std::string         m_tablename;
  int                 m_id_1;
  std::stringstream   m_ss;
  std::stringstream   m_slice;
};

std::string BusinessAdapter::insertRecord(
  const BusinessRecord& r
)
{
  m_ss.str("");
  m_ss << "insert[" << m_tablename << ";(";
  m_ss << makeSlice(r.member_1)    << ";";
  m_ss << makeSlice(r.member_2)    << ";";
  m_ss << makeSlice(r.member_3)    << ";";
  m_ss << makeSlice(r.member_4)    << ";";
  m_ss << makeSlice(m_id_1)        << ";";
  m_ss << makeSlice(r.member_5)    << ";";
  . . . // same for members 6 => 18
  m_ss << makeSlice(r.member_19)   << ")]";
  return m_ss.str();
}
```

Listing 9

```
std::string BusinessAdapter::makeSlice(
  const int val)
{
  m_slice.str("");
  if ( val == intNaN )
    m_slice << "0N";
  else
    m_slice << val;
  return m_slice.str();
}
```

The client is a heavy (and happy) user of a customised version of Pantheios [PAN], and wished to know whether there were similar performance gains to be had in a more general way in the manipulation of strings. This was at a time before FastFormat had been released (and this was a primary impetus to my getting it out there), and I was able to show them the performance speed-up shown in Table 4; in this case the times are measured in microseconds for 10,000 invocations (since there's no I/O).

The two FastFormat implementations are shown in Listings 10 and 11. Neither can be said to be significantly more transparent than the original. Actually, in this case I'd concede that **`FastFormat.Format`** is less transparent, simply because having 21 replacement parameters in a format string is a challenge to maintenance. None of the solutions are easy on the eye, probably because a string is being built from 21 arguments. The primary discriminant in this case is performance. I would observe that at least the FastFormat.Format version has an extra level of error-checking over the other two, since if there are too many or too few arguments, an exception will be thrown.

- Listing 10 shows a **`FastFormat.Format`** implementation of the Database Inserter Statement scenario
- Listing 11 shows a **`FastFormat.Write`** implementation of the Database Inserter Statement scenario

Readers of part 2 [FF2] should recognise aspects of the implementation of the **`make_slice()`** inserter function (Listing 12), which provides equivalent semantics to the original **`makeSlice()`** but uses shim strings [XSTLv1, FF2] to avoid memory allocations, and to reuse the existing FastFormat integer-to-string conversions.

The number of memory allocations have been reduced from five to one

```
std::string BusinessAdapter::insertRecord(
  const BusinessRecord& r
)
{
  std::string result;
  ff::fmt(
    result
  ,
"insert[{0};({1};{2};{3};{4};{5};{6};{7};{8};{9};
{10};{11};{12};{13};{14};{15};{16};{17};{18};{19}
;{20})]"
  ,   m_tablename
  ,   make_slice(r.member_1)
  ,   make_slice(r.member_2)
  ,   make_slice(r.member_3)
  ,   make_slice(r.member_4)
  ,   make_slice(m_id_1)
  ,   make_slice(r.member_5)
  . . . // same for members 6 => 18
  ,   make_slice(r.member_19)
  );
  return result;
}
```

Listing 10

```
std::string BusinessAdapter::insertRecord(
  const BusinessRecord& r
)
{
  std::string result;
  ff::write(
      result
  ,   "insert[",  m_tablename
  ,   ";(",       make_slice(r.member_1)
  ,   ";",        make_slice(r.member_2)
  ,   ";",        make_slice(r.member_3)
  ,   ";",        make_slice(r.member_4)
  ,   ";",        make_slice(m_id_1)
  ,   ";",        make_slice(r.member_5)
  . . . // same for members 6 => 18
  ,   ";",        make_slice(r.member_19)
  ,   ")]"
  );
  return result;
}
```

Listing 11

CComBSTR, std::string, std::wstring and CString

This example is heavily edited from a client's codebase. Please don't try to understand the functions' original purposes, just focus on the

```
stlsoft::basic_shim_string<char, 20>
  make_slice(const int val)
{
  if(intNaN == val)
  {
    return stlsoft::basic_shim_string<char,
       20>("0N");
  }
  else
  {
    return fastformat::filters::filter_type(val,
       &val, static_cast<char const*>(0));
  }
}
```

Listing 12

formulation of the result string. Listing 13 shows the old version of **`GetFilter()`** (along with some other things, representative of the original code, that are required just to get the snippet to compile)

The comments indicate the number of memory allocations, involved in a typical invocation. The type transitions in this case are **`std::string`** -> **`CString`** (x2), **`wchar_t const*`** -> **`CString`**, and **`CString`** -> **`CComBSTR`**. Listing 14 shows the new version of **`GetFilter()`** that is implemented in terms of **`FastFormat.Write`**, using the sink for **`CComBSTR`**.

As you can see, the original five statements have been reduced to two, and the number of memory allocations have been reduced from five to one. Were it not for the need to compress the main statement into the narrow display confines of this magazine, it would be evident that it's also more transparent than the original. There's no performance test for this case – the main aim in the change was to increase transparency and maintainability, and aiding in the project-wide task of removing dependency on the MFC library (incl. **`CString`**).

MessageBox

This last example, another extract from a commercial project, is also about improvements to transparency. The original code in this case was far too big to include here, and I really didn't want to have to think up all the obfuscated names. Furthermore, it did not have the same level of functionality, so only the new version is shown (Listing 15). The code comes from a Windows GUI application that needs to process files and, as shown, report to the user if the file cannot be accessed. For localisation purposes, message strings, and windows error strings, are obtained at runtime.

Assume **`fileName`** is **`"abc.def"`**, and that no such file exists. Further assume that the module designated by **`hinst`** has a string resource with the identifier **`IDS_FMT_MISSING_FILE`** whose value is

```
"The file '{0}' could not be processed: {2}"
```

This class represents a façade over the Windows Resources API functions

```
#define atFilter            1
#define strFilterSeparator  L"-"
enum dimension_t;
HRESULT XXGetFilterEx_(
  dimension_t dimension
, int newIndex
, std::string* fg
, std::string* fv
); // Assign to *fg and *fv
int offset_to_new_index_(
  int filter
, dimension_t dimension
, int index
); // calc index, e.g. 'return index + 1;'
HRESULT GetFilter(
  dimension_t dimension
, short index
, BSTR* filter
)
{
  std::string  szFG;
  std::string  szFV;
  int  newIndex  = offset_to_new_index_(
     atFilter, dimension, index);
  XXGetFilterEx_(dimension, newIndex,
     &szFG, &szFV);
  CString flt(szFG.c_str());  // +1
  flt += strFilterSeparator;  // +2
  flt += szFV.c_str();        // +1
  CComBSTR filter(flt);       // +1
  *filter = filter.Detach();
  return S_OK;
}
```

Listing 13

```
HRESULT GetFilter(dimension_t dimension, short
index, BSTR* filter)
{
  . . . // as before
  CComBSTR filter;
  *filter = fastformat::write(
    filter
  , szFG
  , winstl::w2m(strFilterSeparator)
  , szFV
  ).Detach(); // +1
  return S_OK;
}
```

Listing 14

```
void ProcessFile(
  HINSTANCE hinst
, HWND      parent
, LPCTSTR   fileName
)
{
  WIN32_FIND_DATA fd;
  HANDLE h = ::FindFirstFile(fileName, &fd);

  if(INVALID_HANDLE_VALUE == h)
  {
    DWORD err = ::GetLastError();
    ff::windows_resource_bundle bundle(hinst);

    ff::ignore_unreferenced_arguments_scope
       scoper;

    ff::fmt(
        ff::sinks::MessageBox(parent, "Problem"
      , MB_ICONWARNING)
      , bundle[IDS_FMT_MISSING_FILE]
      , filename
      , err
      , winstl::error_desc(err));
  }
  else
  {
    // . . . do something useful with file data
  }
}
```

Listing 15

In that case, a message box will be displayed, as a child of the parent window, with the type **`MB_ICONWARNING`** (the yellow triangle with an exclamation mark in it), and the message

`"The file 'abc.def' could not be processed: The system cannot find the file specified"`

Obviously, there's a lot going on here: formatting, looking up resources, looking up error code strings. Let's break it down according to the separate FastFormat components at play.

First, an instance of the **`fastformat::windows_resource_bundle`** class is declared, taking **`hinst`** in its constructor. This class represents a façade over the Windows Resources API functions, providing a simple mapping of id to string, throwing an exception if a given id does not represent a string resource.

Second, the creator function **`fastformat::sinks::MessageBox()`** [XSTLv1] constructs an instance of the sink class **`fastformat::sinks::MessageBox_sink`**, which will receive the formatted statement results and then invoke the Windows function

We want to be able to use different resource strings without breaking the application

`MessageBox()` to display the message. In this case, the two arguments required are the filename (`{0}`) and a temporary instance of the `winstl::error_desc` class (`{2}`), which is used to elicit the string form of an error code via the Windows `FormatMessage()` function. Note that we remember the error code associated with the failure to open/stat the file before doing any error display processing, since any subsequent Windows API failure would change the thread's last error code, and lead to a potentially misleading cause being presented to the user.

Third, you may have noticed that there are actually three format arguments: `filename`, `err`, and `winstl::error_desc(err)`, but the example format string contains just two replacement parameters. By default, all format specification defect conditions result in the throwing of an exception (derived from `fastformat::fastformat_exception`). The purpose of the scoper instance of the succinctly named `fastformat::ignore_unreferenced_arguments_scope` class is to suppress this, and allow the string to be formatted despite the mismatch. During its lifetime it suppresses the throwing of a `fastformat::unreferenced_argument_exception`. (We'll discuss how this works later.)

We do this because we want to be able to use different resource strings without breaking the application. This is usually for localisation purposes, but may also be to give more information in debug builds (such as the numeric value of the error code `err`). For example, for some locales we might want to change `IDS_FMT_MISSING_FILE` to:

```
"The file '{0}' could not be processed"
```

And in debug builds we might want to use the format string:

```
"The file '{0}' could not be processed: error code
{1}: {2}"
```

Note that scoper only suppresses exceptions with unreferenced arguments, however. If the format changed to

```
"The file '{0}' could not be processed: please
inform {3}"
```

then a `fastformat::missing_argument_exception` would be thrown, and we don't want to squash that. (If we did, we'd declare an instance of `fastformat::ignore_missing_arguments_scope`, with the effect that `{3}` would be replaced with the empty string.)

Windows format strings

One last note on format strings on Windows. Windows message files (used via `FormatMessage`) and MFC (used via `AfxFormatString*()`) use a different format syntax, where the format string would instead be

```
"The file '%1' could not be processed: %3"
```

As a convenience, the `windows_resource_bundle` class is able to use these format strings, and performs a translation if the original contains one or more Windows replacement parameters and zero FastFormat replacement parameters. This allows an easy upgrade from MFC-based resource formatting to FastFormat, which is particularly useful if your application is localised to several locales.

Format specification defect handlers

Let's now consider the format specification defect handler mechanism. As discussed in part 1 [FF1], format specification defects involve both badly formed format strings, and a failure to match all replacement parameters to all given arguments. We'll consider only the mismatch case; the two aspects follow the same pattern.

Control of mismatch behaviour involves an enumeration, a handler function prototype, a structure, and four API functions, to get+set the handler for thread+process. Thread handlers, if set, take precedence over process ones; in single-threaded builds they are the same.

The relevant aspects of the API are shown in Listing 16. (I apologise for the long names.)

With this, as shown in Listing 17, we're in a position to see how the `fastformat::ignore_unreferenced_arguments_scope` class works.

The class is derived (privately) from the class `fastformat::mismatched_arguments_scope_base`, which handles (de-)registration of a derived class instance and its handler method for the duration of its lifetime, via the class

```
enum ff_replacement_code_t
{
    FF_REPLACEMENTCODE_SUCCESS = 0
  , FF_REPLACEMENTCODE_MISSING_ARGUMENT
  , FF_REPLACEMENTCODE_UNREFERENCED_ARGUMENT
};
typedef int (*fastformat_mismatchedHandler_t)(
  void*                 param
, ff_replacement_code_t code
, size_t                numParameters
, int                   parameterIndex
, ff_string_slice_t*    slice
, void*                 reserved0
, size_t                reserved1
, void*                 reserved2
);
struct ff_mismatched_handler_info_t
{
  fastformat_mismatchedHandler_t handler;
  void* param;
};
ff_mismatched_handler_info_t
  fastformat_getThreadMismatchedHandler();
ff_mismatched_handler_info_t
  fastformat_setProcessMismatchedHandler(
    fastformat_mismatchedHandler_t handler
  , void* param
  );
. . . // same for process handlers
```

Listing 16

Note that the gains, if any, are strongly platform-dependent

fastformat_setThreadMismatchedHandler(). (The use of **get_this_()** is an old trick for avoiding compiler warnings about use of a partially constructed instance in its own member initialiser list; see [IC++].)

```
// in namespace fastformat
class ignore_unreferenced_arguments_scope
  : private mismatched_arguments_scope_base
{
public:
  typedef ignore_unreferenced_arguments_scope
     class_type;
  typedef mismatched_arguments_scope_base
     parent_class_type;
public:
  ignore_unreferenced_arguments_scope()
    : parent_class_type(class_type::handler,
    get_this_())
  {}
private:
  void* get_this_() throw()
  {
    return this;
  }
  static int handler(
    void*                 param
  , ff_replacement_code_t code
  , size_t                numParameters
  , int                   parameterIndex
  , ff_string_slice_t*    slice
  , void*                 reserved0
  , size_t                reserved1
  , void*                 reserved2
  )
  {
    class_type* pThis =
       static_cast<class_type*>(param);
    if(FF_REPLACEMENTCODE_UNREFERENCED_ARGUMENT
       == code)
    {
      return +1; // Ignore unreferenced argument
    }
    else
    {
      return pThis->parent_class_type::
         handle_default(param, code,
         numParameters, parameterIndex, slice,
         reserved0, reserved1, reserved2);
      }
  }
}
```

Listing 17

The meat of this component is in the derived class's **handler()** method, which intercepts an unreferenced argument code, and instructs the FastFormat replacement engine to ignore it. All other codes are passed, via the parent class's **handle_default()**, to the previous handler, if any, in the chain. The consequence of this is that, for the lifetime of scoper, any unreferenced arguments will not cause an **fastformat::unreferenced_argument_exception** to be thrown.

You're not limited to the scoping classes provided with the distribution. FastFormat allows you to customise its behaviour in light of ill-formed format strings and/or of mismatched arguments, on a per-process and/or per-thread basis, to do whatever funky things your heart desires.

Choosing output sinks

A last note on output sinks. If you're determined to squeeze out every last cycle, you might choose to use **stdout** as your sink rather than **std::cout**, as in:

```
#include <fastformat/sinks/FILE.hpp>
//#include <fastformat/sinks/ostream.hpp>

FILE* stm = stdout;

// FastFormat.Format
ff::fmtln(
  stm
, "[{0}{1}]"
, ff::to_f(12345.12345, -14, 3)
, ff::to_f(12345.12345, -14, 3)
);
```

The inconvenience with this is that you'll likely have to declare a sink variable, as shown in the example, because **stdout** is often not an instance at all, but rather some **#define** into part of an implementation-defined structure, such as **(&_iob[1])** for Visual C++, or **(&_streams[1])** with Borland. The same is needed for **stderr**.

Note that the gains, if any, are strongly platform-dependent. On Linux it gives between 5% and 10% improvement, on Mac OS-X between 2% and 6%. On Windows, with VC++ 9 it actually slows down by a small margn. Obviously, the advice is to get it working (with **std::cout**) and then optimise (with **stdout**) if you really need to. That's only likely to be if you're writing to a file – using an arbitrary **FILE*** handle rather than an arbitrary **fstream** instance – since console/terminal output is far too much affected by other factors for such low-percentage performance improvements to be significant.

FastFormat for logging?

Several people have enquired about the use of FastFormat for application logging. As we've seen in the Server Connection Log example, for some kinds of logging it's a good solution. For what I call application logging, however – the presence of statements throughout the code that allows an interested observer to follow what is happening now, and what has already

happened, at a high level of granularity – the use of **FastFormat.Format**, or any other replacement-based API [FF1], is not advisable. The same goes for any library that is less than 100% type-safe. The reason is that the programmer should be able to have full confidence that an application logging statement will be processed and emit output. This is because many uses of log statements are in places in the code are impossible, or exceedingly difficult, to test, and usually these statements are the ones you most need to be able to rely on.

I've mentioned my other, older, logging API library, Pantheios, a couple of times in this article series. I hope to write an article about that at some time in the future, and will go into more detail about the how/why/when/what of logging. For the moment, however, I'll show you a sneaky trick that allows you to use **FastFormat.Write** (or **FastFormat.Format**, if you must) with Pantheios.

The Pantheios application layer contains severity level pseudo-constant symbols that are actually stateless global instances of specialisations of a severity level class template:

```
namespace Pantheios
{
  namespace
  {
    static level<PANTHEIOS_SEV_DEBUG> debug;
    . . . // and so on
    static level<SEV_ALERT> alert;
    static level<SEV_EMERGENCY> emergency;
  }
}
```

These are not declared **const**, even though no-one should be attempting any mutations of them, precisely so they can be used as 'sinks' to FastFormat, by defining the following overload of **fastformat::sinks::fmt_slices** [FF2]:

This 'works' because the definitions of **ff_string_slice_t** and **pan_slice_t** – the thing **ff_string_slice_t** was copied from in the first place – are identical, and therefore binary compatible. The two static asserts [IC++] are there to ensure that any changes that invalidate that assumption are not missed. Then it's as simple as casting from one array of string alices to the other, and passing to the core Pantheios logging function.

This allows code such as the following:

```
catch(std::exception& x)
{
  ff::write(pan::warning,
     "Something bad has happened: ", x);
}
```

and

```
HRESULT GetFilter(dimension_t dimension,
    short index, BSTR* filter)
{
  // NOTE: can pass 'dimension' variable
  // directly if string access shims are
  // defined for the dimension_t enum
  ff::write(pan::debug, "GetFilter(dimension=",
     dimension, "; index=", index, ", ...)");
```

As I said at the start of this section, there are several reasons to prefer using a proper logging solution, such as Pantheios, but this technique will get you a fair way along to a good solution.

```
// in namespace fastformat::sinks
template <int L>
pantheios::level<L>& fmt_slices(
  pantheios::level<L>&     sink
, int                      /* flags */
, size_t                   /* cchTotal */
, size_t                   numResults
, ff_string_slice_t const* results)
{
  STLSOFT_STATIC_ASSERT(sizeof(
     pantheios::pan_slice_t) ==
     sizeof(fastformat::ff_string_slice_t));
  STLSOFT_STATIC_ASSERT(offsetof(
     pantheios::pan_slice_t, len) ==
     offsetof(fastformat::ff_string_slice_t,
     len));

  pantheios::pantheios_log_n(
    sink
  , numResults
  , stlsoft::sap_cast<pantheios::
     pan_slice_t const*>(results)
  );
  return sink;
}
```

Listing 18

Summary

This article completes the introduction to FastFormat, a C++ library that applies advanced generic conversion techniques to provide robust, flexible and efficient formatting, providing answers to the deficiencies of the current standard and widely used third-party libraries. It is in ongoing development, and readers are invited to use, criticise and contribute, as they see fit. ■

References

[FF1] 'An Introduction to FastFormat, part 1: The State of the Art', Matthew Wilson, *Overload* #89, February 2009; http://accu.org/index.php/journals/c249/

[FF2] 'An Introduction to FastFormat, part 2: Custom Argument and Sink Types', Matthew Wilson, *Overload* #90, April 2009; http://accu.org/index.php/journals/c251/

[IC++] *Imperfect C++*, Matthew Wilson, Addison-Wesley 2004; http://www.imperfectcplusplus.com/

[PAN] The Pantheios Logging API Library, http://www.pantheios.org/; to see why it's the best choice in C++ logging APIs, check out http://www.pantheios.org/performance.html#sweet-spot, which shows graphically how Pantheios can be up to two-orders of magnitude faster than the rest.

[PragProg] *The Pragmatic Programmer*, Dave Thomas and Andy Hunt, Addison-Wesley, 2000; http://www.pragmaticbookshelf.com/

[SO] http://www.stackoverflow.com/questions/586410/

[XSTLv1] *Extended STL*, volume 1, Matthew Wilson, Addison-Wesley 2007; http://www.extendedstl.com/

The Model Student: The Enigma Challenge

Codebreaking was instrumental to computing history. Richard Harris presents a simplified Enigma code for you to crack.

I was fortunate enough to be invited to assist in the ACCU's charitable efforts for Bletchley Park at this year's conference by creating a cryptographic puzzle that would form part of the fundraising effort. It was intended to be simpler to solve by hand than by computer, and to encourage the former included a question that could not be answered if one simply wrote a program to examine every potential solution. In fact, it was designed so that it should be possible to solve the challenge by hand with pen and paper in about 15 to 30 minutes, albeit only if one had spotted the properties of the puzzle that made such a speedy solution possible.

So that you too might enjoy the challenge it is presented below, followed by an historical justification and then its solution.

The challenge

Encoding

The enemy are using a 36 character alphabet encoded with 2 digit base 6 numbers such that the numerical value of the character is equal to 6 times the first digit plus the second digit. The # character is a control code indicating that the following two digits should be interpreted as a two digit base 6 integer in the range 0 to 35, rather than as a letter or punctuation. The full table of character mappings is given below:

```
000000111111222222333333444444555555
012345012345012345012345012345012345
#abcdefghijklmnopqrstuvwxyz ?!.+-*/=
```

Encryption

The encryption scheme used by the enemy is an Enigma variant with two 6 digit rotors of the form shown in Figure 1.

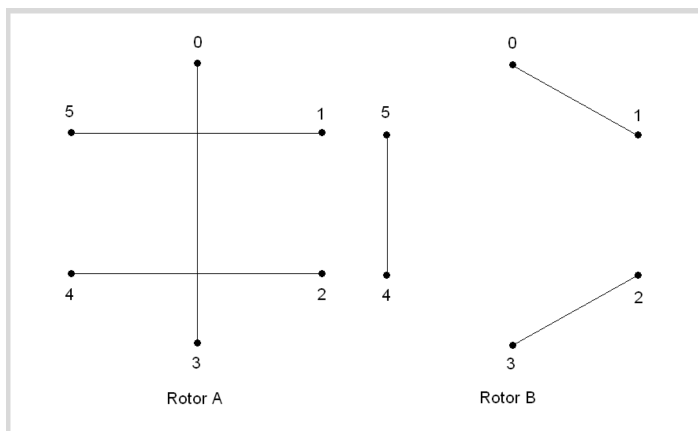


Figure 1

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

When a plaintext digit is entered it is mapped to an intermediate value by the first rotor (which may be either A or B) and then to the cyphertext by the second rotor along the connecting lines. For example, given the setting above, where rotor A is the first and rotor B the second, the digit 1 would be mapped by rotor A to an intermediate value of 5 and then by rotor B to the cyphertext digit 4.

After each digit (i.e. half a character) is entered the connections on the second rotor rotate 1 digit clockwise. After every 6 digits, the connections on the first rotor also rotate 1 digit clockwise.

The order and orientation of the initial setting of the rotors is unknown.

Plaintext

We know that the enemy prefix their plaintext messages with a single base 6 digit (i.e. half a character) representing the message's priority. We also know that they foolishly sign the plaintext so that the last characters are always of the form:

```
NN...NNDDMMM
```

where N stands for the surname of the agent, D for the day and M for the month. For example:

```
harris04feb
```

Cyphertext

Decrypting the following cyphertext will reveal which drop to deposit your ticket in:

```
514524110354101255444144222503405
```

Bonus question

Defining *use* to mean 'consider the implications for more than any two digits (i.e. one character's worth of data) in the cyphertext', how many initial rotor settings must we use in the worst case to guarantee that we correctly decrypt this message?

The historical justification

Given that Bletchley Park was the intended beneficiary of our charitable efforts it was a natural choice to base the puzzle on the Enigma machine, an example of which is illustrated in Figure 1. The difficult part was ensuring that the weaknesses I introduced into the puzzle reflected historic weaknesses in the design and use of the Enigma machine.

Originally marketed to businesses as a secure means of corporate communication, the Enigma machine was adopted by the German armed forces between the years of 1926 and 1935; the Navy were the first to adopt it and the Air Force the last [Copeland04].

The original machine had three rotors around the edge of which were printed the 26 letters of the alphabet. They could be placed in any order in the machine and were internally wired to create electrical paths connecting pairs of letters. Figure 2 shows an Enigma machine (photo by K. Sperling).

Adjacent to the left-most rotor was a reflector board connecting pairs of letters on the left-most rotor, resulting for each letter in a path leading

The mechanism of the Enigma machine ensured that the substitution rule changed after every letter in the message

Figure 2

through the assembly from the right-most rotor to the left-most, through the reflector board and back again from the left-most to the right-most, conceptually illustrated in Figure 3.

After each key press the right-most rotor would rotate by one step. After it had rotated through all 26 steps the middle rotor would rotate by one step and after it, in turn, had rotated through all 26 steps the left-most rotor would rotate by one (the rotation of the rotors was, in fact, slightly more complicated than this, but this description captures the basic idea).

Finally, the keys used to input the message and the lamps lighting up the encrypted letters were wired to the right-most rotor such that the encryption was symmetric; if the key Q resulted in the letter A lighting up for a given set-up, the key A would result in the letter Q lighting up, as illustrated in Figure 4.

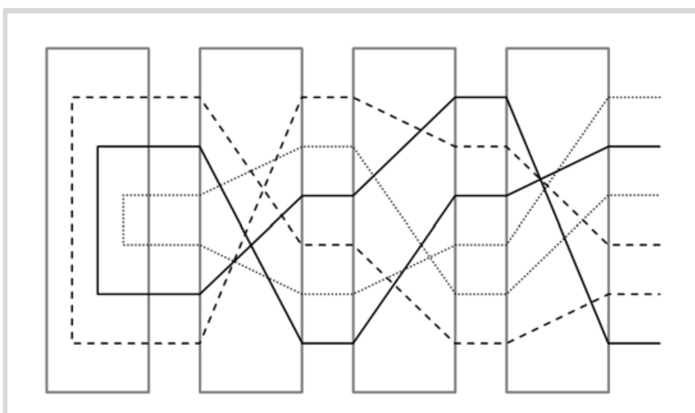


Figure 3

Hence to decrypt a message, one needed simply to type it into an Enigma machine configured in exactly the same way as the one that was used to encrypt it.

The encryption scheme resulting from this mechanism is a known as a substitution cipher, in which each letter in the message is encrypted by substituting it with another. When the substitution rule is fixed for the whole message, this scheme is fairly easy to crack. The mechanism of the Enigma machine ensured that the substitution rule changed after every letter in the message and only cycled through all of the rules a given set-up could represent after $26 \times 26 \times 26 \approx 17,500$ key presses. Furthermore, since the 3 rotors could be arranged in 6 different ways, there were 6 times as many initial settings and hence sequences of substitution rules.

The military version of the Enigma machine added further complexities to increase first the number of initial settings and then the length of the cycles of substitution rules.

The encryption scheme described in the Enigma challenge is clearly a much simplified version of the Enigma machine. Whilst it follows the same basic principle of implementing a substitution cipher using connections between letters on rotors, and of changing the substitution rule after each letter in the message by rotating them, it only has a 6 letter alphabet, 2 rotors and it entirely does away with the reflector board.

Now that I have described the mechanism of the Enigma machine and how the challenge is just a simplified version of the encryption scheme it implemented, it's time to describe a couple of its weaknesses which inspired the specific details of the challenge that make it relatively simple to solve.

One weakness of the Enigma machine was the order in which the rotors were rotated after each key press; if the middle rotor had been the first to rotate rather than the right-most it would have been considerably more difficult to crack [Leavitt06].

To reflect this weakness, the Enigma challenge uses deliberately badly designed rotors; the specific details are given in the solution, below.

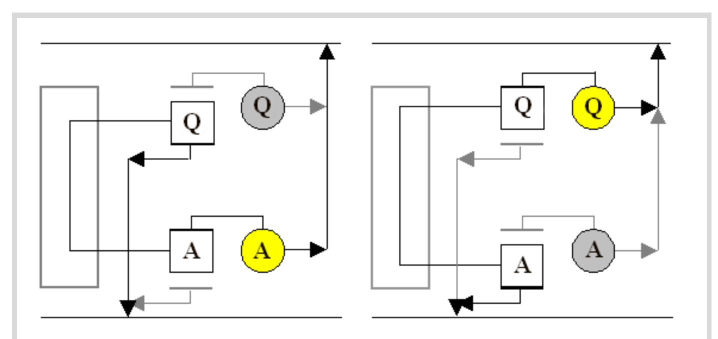


Figure 4

The next weakness of the Enigma machine, at least in the early days of its use by the German military, was that rather than use a daily pre-distributed setting for encoding messages, they would use such a daily setting to encode a 3 character message defining the rotor setting for the remainder of the message. Whilst on the face of it this seems to be a pretty sensible scheme, they further decided to repeat the message setting twice to allow checking for errors in the settings or the message transmission. By enforcing that the first 6 characters of the message were in fact a pair of identical 3 character messages, the German military had provided their enemies with a hint, or crib, as to what the daily rotor settings were.

This is reflected in the Enigma challenge by the message signature. By consistently signing their messages in a specific format, our hypothetical enemies have introduced a fatal weakness into the encryption scheme, as detailed in the solution, below.

The solution

To decrypt this message, the first thing to note is the rotational symmetries of the rotors, meaning that there are only 6 unique settings for each ordering. The second thing to note is that reversing the order of the rotors reverses the mapping of the digits, meaning that we can deduce the mapping for both orderings by examining only one.

Naming the rotor positions as in their initial description and rotated clockwise through their distinct states as A_0, A_1, A_2, B_0 and B_1 respectively and using permutation notation (in which each number in the top rows are encrypted to the numbers below them) we have:

$$A_0B_0 = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 2 & 4 & 5 & 1 & 3 & 0 \end{pmatrix} \Rightarrow B_0A_0 = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 5 & 3 & 0 & 4 & 1 & 2 \end{pmatrix}$$

$$A_0B_1 = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 4 & 0 & 3 & 5 & 1 & 2 \end{pmatrix} \Rightarrow B_1A_0 = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 1 & 4 & 5 & 2 & 0 & 3 \end{pmatrix}$$

$$A_1B_0 = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 3 & 5 & 1 & 4 & 0 & 2 \end{pmatrix} \Rightarrow B_0A_1 = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 4 & 2 & 5 & 0 & 3 & 1 \end{pmatrix}$$

$$A_1B_1 = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 1 & 3 & 5 & 0 & 2 & 4 \end{pmatrix} \Rightarrow B_1A_1 = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 3 & 0 & 4 & 1 & 5 & 2 \end{pmatrix}$$

$$A_2B_0 = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 5 & 2 & 4 & 0 & 1 & 3 \end{pmatrix} \Rightarrow B_0\mathrm{A}_2 = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 3 & 4 & 1 & 5 & 2 & 0 \end{pmatrix}$$

$$A_2B_1 = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 3 & 4 & 0 & 2 & 5 & 1 \end{pmatrix} \Rightarrow B_1\mathrm{A}_2 = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 2 & 5 & 3 & 0 & 1 & 4 \end{pmatrix}$$

Next we note that the day in the signature will be an integer between 0 and 31, so will be encoded in the number format; namely two 0 digits followed by the value in base 36. This means that the 9th and 10th digits from the end of the encoded plaintext must both be 0. Hence, at that point in the encryption we have:

$$0 \to 4 \text{ then } 0 \to 2$$

This could only occur with the sequential rotor positions A_0B_1, A_0B_0 or B_0A_1, B_1A_2.

Finally we note that the 10th digit from the end is the 24th from the start and, since this is a multiple of 6, both rotors must turn after it is encrypted. Hence the rotor positions at this point must be B_0A_1. Furthermore, since 24 is an even multiple of 6 and rotor B only has 2 unique states, the next state must be equivalent to the starting position of the rotors, namely B_1A_2 and so the rotors must cycle through the sequence:

$$B_1A_2, B_1A_0, B_1A_1, B_1A_2, B_1A_0, B_1A_1, B_0A_2, B_0A_0, B_0A_1, B_0A_2, B_0A_0, B_0A_1$$

Applying this to the cyphertext gives us:

```
102132243000443024130230035140122
```

Hence the plaintext is:

```
1bin #04 byro#35jan
```

or:

```
(priority 1) bin 4 byro23jan
```

and we only need use *1* initial rotor setting to successfully decrypt the message.

In closing

Well, I hope you enjoyed the Enigma challenge and that it gave you a taste of the technical process of code breaking, or cryptanalysis. If this has piqued your interest in the subject and you would like to read more about it, Simon Singh's *The Code Book* [Singh99] provides an enjoyable layman's treatment. For the more mathematically minded, there are many textbooks on the subject: *Handbook of Applied Cryptography* [Menezes97] is the one currently sitting on my desk.

I would also recommend a visit to Bletchley Park itself with its many exhibits on the code breaking work undertaken there during the Second World War and its fledgling Computer Museum [Bletchley]. Whilst we're about it, why not give them a donation? As a fundamental part of our collective history as computer programmers and users, we owe it to ourselves to see that this site is maintained for future generations.

This article has been something of a departure from our usual subject matter, but given the importance of the beneficiary, I believe it to be a justifiable one.

Next time, dear reader, normal service will be resumed. ■

And finally...

Congratulations to Jim Hague, Alan Brooks, Phil Bitis, Andrew Bainbridge, Sam Saariste, Roger Orr, Per Liboriussen, Dave Hargreaves, Jan Willem, Jonathan Wakely, Charles Tolman, Judy Booth, Seb Rose, Jakob Gaardsted, George Vernon, Gary Duke, who all broke the code, and especially to Duncan Grant, Matthias Hertel and Andrew Kemp who successfully cryptanalysed the Enigma Challenge.

To everyone who took part, and to everyone who donated to Bletchley, we should like to extend our deepest gratitude.

Acknowledgements

With thanks to Astrid Byro, John Paul Barjaktarevi? and Lee Jackson for proof reading this article.

References and further reading

[Bletchley] www.bletchleypark.org.uk

[Copeland04] Copeland, B. (ed), *The Essential Turing*, Oxford University Press, 2004

[Leavitt06] Leavitt, D., *The Man Who Knew Too Much*, Weidenfeld & Nicolson, 2006

[Menezes97] Menezes, A. et al, *Handbook of Applied Cryptography*, CRC Press, 1997

[Singh99] Singh, S., *The Code Book*, Fourth Estate, 1999

Boiler Plating Database Resource Cleanup (Part 2)

Timely disposal of resources is important. Paul Grenyer applies this to database access in Java.

In my recent CVu [CVu] article, 'Boiler Plating Database Resource Cleanup – Part I' [Part I] I explained that cleaning up after querying a database in Java is unnecessarily verbose and complex and demonstrated how boiler plate code could be developed to reduce the amount of client code needed using the FINALLY FOR EACH RELEASE pattern [AToTP]. In this article I am going to look at an alternative boiler plate solution using the EXECUTE AROUND METHOD (EAM) [AToTP] pattern. But first, let's take another brief look at the problem.

The problem – revisited

The problem is simple. Cleaning up after querying a database in Java is unnecessarily verbose and complex. Plain and simple. Listing 1 is the code needed to lookup a single string in a database.

This is a lot of code to get one string out of a database and most of it must be repeated every time a database is accessed. Most of it is error handling and resource management. For a more detailed look at this code see Part I.

Execute Around Method

The EAM pattern is described by Kevlin Henney in his article 'Another Tale of Two Patterns'. EAM describes how to 'encapsulate pairs of actions in the object that requires them, not code that uses the object, and pass usage code to the object as another object'.

The advantage of EAM over FINALLY FOR EACH RELEASE is that the client is able to use a resource simply by implementing an interface, using the resource passed to the subclass without worrying about how to clean up and then simply pass an instance of the subclass to another object for resource acquisition, execution and cleanup.

To check or not to check

Checked **Exception**s [CheckedExceptions] in Java have their advantages and disadvantages and are a source of much controversy. In my previous article I made all of my interface methods throw **Exception** (except for **ConnectionPolicy**, which throws its own custom

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

Listing 1

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

I am forcing client code to catch and deal with checked exceptions or translate and rethrow them as runtime exceptions

```
public interface ErrorPolicy
{
  void handleError(Exception ex);
  void handleCleanupError(Exception ex);
}
```

Listing 2

```
public class DefaultErrorPolicy
   implements ErrorPolicy
{
  private Exception firstException = null;
  @Override
  public void handleCleanupError(Exception ex)
  {
    handleError(ex);
  }
  @Override
  public void handleError(Exception ex)
  {
    if (firstException == null)
    {
      firstException = ex;
      throw new RuntimeException(ex);
    }
  }
}
```

Listing 3

exception) so that client code can throw almost any exception type it likes. This effectively negates the checked part of checked exceptions.

Instead of using checked exceptions for the EAM design, I am forcing client code to catch and deal with checked exceptions or translate and rethrow them as runtime exceptions, by omitting any exception specification from interface method signatures.

To aid with this I have written an `ErrorPolicy` interface (Listing 2) and a `DefaultErrorPolicy` class (Listing 3) that translate `Exception` into `RuntimeException` where appropriate.

The `ErrorPolicy` interface is designed to allow exceptions thrown as a result of an error from using a database resource to be handled differently to those thrown as a result of cleaning up a database resource. For example, a user may want to rethrow only use exceptions and simply log or ignore cleanup exceptions.

`DefaultErrorPolicy` only rethrows the first exception it is asked to handle. This guarantees that a cleanup exception, which would generally be thrown after a use exception, does not hide the use exception. If I was including the ability to log in my design I would log all exceptions handled by `DefaultErrorPolicy`.

The error policy must always be set and *can* always be used in the same way. To provide the necessary consistency when setting the error policy I wrote the following interface:

```
public interface ErrorPolicyUser
{
  void setErrorPolicy(ErrorPolicy errorPolicy);
}
```

To provide a common, optional method of storing and accessing a reference to the error policy I wrote the abstract class in Listing 4.

From policy to factory

The more I thought about and discussed the `ConnectionPolicy` interface from my previous article, the more I felt it was more like an Abstract Factory [GoF] than a policy. Therefore I have renamed it.

```
public interface ConnectionFactory
   extends ErrorPolicyUser
{
  Connection connect();
  void disconnect(Connection con);
}
```

I want `ConnectionFactory` clients to be forced to accept an error policy without relying on it being passed to subclass constructors the interface has no control over. Extending the `ErrorPolicyUser` interface also means that the error policy can be set internally by `ConnectionProvider` (discussed next).

Most `Connection` objects are cleaned up in the same way, so having the common code encapsulated in an abstract class prevents unnecessary code duplication. An abstract class is also the ideal place for boiler plate error policy handling (Listing 5).

`AbstractConnectionFactory` is a good example of how a class that needs to implement `ErrorPolicyUser` can extend

```
public abstract class AbstractErrorPolicyUser
   implements ErrorPolicyUser
{
  private ErrorPolicy errorPolicy =
     new DefaultErrorPolicy();
  protected ErrorPolicy getErrorPolicy()
  {
    return errorPolicy;
  }
  @Override
  public void setErrorPolicy(
     ErrorPolicy errorPolicy)
  {
    this.errorPolicy = errorPolicy;
  }
}
```

Listing 4

```
public abstract class AbstractConnectionFactory
   extends AbstractErrorPolicyUser implements
ConnectionFactory
{
  @Override
  public void disconnect(Connection con)
  {
    if (con != null)
    {
      try
      {
        con.close();
      }
      catch(final SQLException ex)
      {
        getErrorPolicy().handleCleanupError(ex);
      }
    }
  }
}
```

Listing 5

`AbstractErrorPolicyUser` to get the implementation and access to the error policy for free.

The disconnect method is also a good example of how the error policy is used to translate a checked exception into something else, in this case a runtime exception. A little more thought needs to be put into the connection creation factories to make sure all exceptions are caught and passed to the error policy (Listing 6).

```
public class StringConnection
   extends AbstractConnectionFactory
{
  ...
  @Override
  public Connection connect()
  {
    Connection con = null;
    try
    {
      Class.forName(driver);
      con = DriverManager.getConnection(
         connectionString, username, password);
      try
      {
        if (database != null)
        {
          con.setCatalog(database);
        }
      }
      catch(SQLException ex)
      {
        try
        {
          getErrorPolicy().handleError(ex);
        }
        finally
        {
          disconnect(con);
        }
      }
    }
    catch(ClassNotFoundException ex)
    {
      getErrorPolicy().handleError(ex);
    }
    catch(SQLException ex)
    {
      getErrorPolicy().handleError(ex);
    }
    return con;
  }
}
```

Listing 6

ConnectionProvider

The concept of a connection provider was suggested to me by Adrian Fagg. The idea is that a single class is responsible for acquiring a connection, providing it to another class for use and releasing it again. This is Execute Around Method!

The advantage over `DbResourceHandler` from my previous article is equal encapsulation while making the client less restricted by what they can do with the connection. It does, however, suffer from the same disadvantage that one method is required for uses of the connection which return values and another for uses that do not.

To allow for this, two connection use interfaces are required. The `ConnectionUser` interface is for uses of the connection that do not return a value:

```
public interface ConnectionUser
   extends ErrorPolicyUser
{
  void use(Connection con);
}
```

The `ConnectionValue` interface is parameterised for the type returned from uses of the connection that return a value:

```
public interface ConnectionValue<T>
   extends ErrorPolicyUser
{
  T fetch(Connection con);
}
```

The construction of the `ConnectionProvider` class is very straight forward. The basic constructor takes a connection factory, creates a `DefaultErrorPolicy` and passes them both to another constructor that stores the references and passes the error policy to the connection factory. This means that clients of the connection provider are free to use the default error policy or provide their own. (Listing 7)

`ConnectionProvider` has two other methods. One that provides a connection to a `ConnectionUser` and the other which provides a connection to a `ConnectionValue` and returns the fetched value (Listing 8).

```
public final class ConnectionProvider
{
  private final ConnectionFactory conFactory;
  private final ErrorPolicy errorPolicy;
  public ConnectionProvider(
     ConnectionFactory conFactory)
  {
    this(conFactory, new DefaultErrorPolicy());
  }
  public ConnectionProvider(
     ConnectionFactory conFactory,
     ErrorPolicy errorPolicy)
  {
    this.conFactory = conFactory;
    this.errorPolicy = errorPolicy;
    this.conFactory.setErrorPolicy(
       this.errorPolicy);
  }
  ...
}
```

Listing 7

```
public final class ConnectionProvider
{
  ...
  public void provideTo( ConnectionUser user )
  {
    user.setErrorPolicy(errorPolicy);
    final Connection con = conFactory.connect();
    try
    {
      user.use(con);
    }
    finally
    {
      conFactory.disconnect(con);
    }
  }
  public <T> T provideTo(
     ConnectionValue<T> fetcher )
  {
    fetcher.setErrorPolicy(errorPolicy);
    final Connection con = conFactory.connect();
    T result = null;
    try
    {
      result = fetcher.fetch(con);
    }
    finally
    {
      conFactory.disconnect(con);
    }
      return result;
  }
}
```

Listing 8

```
class User extends AbstractErrorPolicyUser
   implements ConnectionUser
{
  @Override
  public void use(Connection con)
  {
    // Use the connection
  }
}
final ConnectionProvider cp =
   new ConnectionProvider(
   new StringConnection(...));
cp.provideTo( new User() );
```

Listing 9

Both methods pass the error policy to the user of the connection, create the connection, pass it to the user and cleanup the connection. They rely on the connection factory to deal with any errors. The example in Listing 9 shows how the **ConnectionProvider** can be used.

The **User** class extends the **AbstractErrorPolicyUser** to get the common error policy storage functionality and implements **ConnectionUser** so that it can be handled by **ConnectionProvider**. There is a single override where the connection is used.

Having to extend **AbstractErrorPolicyUser** *and* implement **ConnectionUser** is not ideal. My original design had an **AbstractConnectionUser** class that extended **AbstractErrorPolicyUser** and implemented **ConnectionUser** so that clients only had to extend a single class. This meant having a similar abstract class for every user and value variant, which did not seem worth it when, as we will see later, the user and value variants are encapsulated in another class unless the client wants something custom.

StatementProvider

The **StatementProvider** class (Listing 10) is a natural progressions from the **ConnectionProvider** and uses EAM in the same way to provide a **Statement** to a client without the client needing to worry about acquisition or cleanup. The construction is simple and takes only a **Connection**, from which to create the statement, and an error policy. Again, it has two **provideTo** methods. One paramatised method that passes the **Statement** to a **StatementValue**:

```
public interface StatementValue<T>
    extends ErrorPolicyUser
{
  T use(Statement stmt);
}
```

and returns a value. The other method passes the **Statement** to a **StatementUser**:

```
public interface StatementUser
    extends ErrorPolicyUser
{
  void use(Statement stmt);
}
```

```
public final class StatementProvider
{
  private final Connection con;
  private final ErrorPolicy errorPolicy;
  public StatementProvider(
     Connection con, ErrorPolicy errorPolicy)
  {
    this.con = con;
    this.errorPolicy = errorPolicy;
  }
  public void provideTo( StatementUser user )
  {
    user.setErrorPolicy(errorPolicy);
    try
    {
      final Statement stmt =
         con.createStatement();
      try
      {
        user.use(stmt);
      }
      finally
      {
        try
        {
          stmt.close();
        }
        catch(SQLException ex)
        {
          errorPolicy.handleCleanupError(ex);
        }
      }
    }
    catch(SQLException ex)
    {
      errorPolicy.handleError(ex);
    }
  }
  public <T> T provideTo(
     StatementValue<T> fetcher )
  {
    fetcher.setErrorPolicy(errorPolicy);
    T result = null;
    try
    {
```

Listing 10

```
      final Statement stmt =
         con.createStatement();
      try
      {
        result = fetcher.use(stmt);
      }
      finally
      {
        try
        {
          stmt.close();
        }
        catch(SQLException ex)
        {
          errorPolicy.handleCleanupError(ex);
        }
      }
    }
    catch(SQLException ex)
    {
      errorPolicy.handleError(ex);
    }
    return result;
  }
}
```

Listing 10 (cont'd)

Both methods pass the error policy to the user of the statement, create the statement, pass it to the user, clean it up again and are responsible for error handling.

The execution of statements that do not return a value is very straight forward, so I wrote `StatementUser` for this purpose (Listing 11).

ResultSetProvider

Executing statements that return one or more values is less straight forward and requires a `ResultSetProvider` (Listing 12).

Again, this is another natural progression from `ConnectionProvider`. The `ResultSetProvider` takes the SQL query to execute to create the result set, a `Statement` from which to create the record set and an error policy. There is only a single paramatised `provideTo` method as a value is always returned.

```
public class Execute
   extends AbstractErrorPolicyUser
   implements StatementUser
{
  private final String sql;
  public Execute(String sql)
  {
    this.sql = sql;
  }
  @Override
  public void use(Statement stmt)
  {
    try
    {
      stmt.execute(sql);
    }
    catch(SQLException ex)
    {
      getErrorPolicy().handleError(ex);
    }
  }
}
```

Listing 11

```
public final class ResultSetProvider
{
  private final String sql;
  private final Statement stmt;
  private final ErrorPolicy errorPolicy;
  public ResultSetProvider(String sql,
     Statement stmt, ErrorPolicy errorPolicy)
  {
    this.sql = sql;
    this.stmt = stmt;
    this.errorPolicy = errorPolicy;
  }

  public <T> T provideTo(
     ResultSetFunction<T> fetcher)
  {
    fetcher.setErrorPolicy(errorPolicy);
    T result = null;
    try
    {
     final ResultSet rs = stmt.executeQuery(sql);
      try
      {
        result = fetcher.read(rs);
      }
      finally
      {
        try
        {
          rs.close();
        }
        catch(Exception ex)
        {
          errorPolicy.handleCleanupError(ex);
        }
      }
    }
    catch(SQLException ex)
    {
      errorPolicy.handleError(ex);
    }
    return result;
  }
}
```

Listing 12

The `ResultSet` is provided to a `ResultSetFunction`:

```
public interface ResultSetFunction<T>
   extends ErrorPolicyUser
{
  T read(ResultSet rs);
}
```

that is parameterised on return type. The method passes the error policy to the user, creates the `RecordSet` from the `Statement` and SQL query, passes the `ResultSet` to the `ResultSetFunction`, cleans up, handles any errors and returns the value.

With the `ResultSetProvider`, executing a query that returns a value is almost as simple as executing one that does not and can benefit from similar boilerplate (Listing 13).

The `ExecuteQuery` class is paramatised on the return type from the query. It takes a SQL query and `ResultSetFunction` via its constructor. When an instance is passed to a `StatementProvider` `provideTo` method it uses a `ResultSetProvider` and the `ResultSetFunction` to execute and return the results of the query.

```
public class ExecuteQuery<T>
   extends AbstractErrorPolicyUser
   implements StatementValue<T>
{
  private final String sql;
  private final ResultSetFunction<T> rsUser;
  public ExecuteQuery(
     String sql, ResultSetFunction<T> rsUser)
  {
    this.rsUser = rsUser;
    this.sql = sql;
  }
  @Override
  public T use(Statement stmt)
  {
    return new ResultSetProvider( sql, stmt,
       getErrorPolicy()).provideTo(rsUser);
  }
}
```

Listing 13

Query

All of this boilerplate can be wrapped in a single class that provides two static methods, one to execute methods that return a value and another for ones that do not (Listing 14).

Both methods take a **ConnectionProvider** and a SQL query. The method which returns a value also takes a **ResultSetFunction** to process the result set into the return value. Both methods pass an instance of a nested class to the connection provider.

The nested class that handles queries that do not return a value is called **User** (Listing 15). It takes the SQL query via its constructor and, when passed to a **ConnectionProvider**, uses the **StatementProvider** and **Execute** classes to execute the query.

The nested class that handles queries that return a value is called **Value** (Listing 16). It takes the SQL query and a **ResultSetFunction** via its constructor and, when passed to a **ConnectionProvider**, uses the **StatementProvider** and **ExecuteQuery** classes and the **ResultSetFunction** interface to execute the query and return the result.

AbstractResultSetFunction

The **ResultSetFunction** interface needs a little further explanation as it is the only interface most clients will need to implement albeit then only for queries that return a value.

```
public final class Query
{
  …
  public static void execute(
     ConnectionProvider conProvider,
     String sql) throws Exception
  {
    conProvider.provideTo(new User(sql));
  }
  public static <T> T execute(
     ConnectionProvider conProvider,
     String sql, ResultSetFunction<T> rsUser)
     throws Exception
    {
      return conProvider.provideTo(
         new Value<T>(sql,rsUser));
    }
  private Query()
  {}
}
```

Listing 14

```
private static class User
   extends AbstractErrorPolicyUser
   implements ConnectionUser
{

  private final String sql;
  public User(String sql)
  {
    this.sql = sql;
  }
  @Override

  public void use(Connection con)
  {
    new StatementProvider(
       con,getErrorPolicy()).provideTo(
       new Execute(sql) );
  }
}
```

Listing 15

```
public interface ResultSetFunction<T>
   extends ErrorPolicyUser
{
  T read(ResultSet rs);
}
```

The interface extends **ErrorPolicyUser**, which means clients can make use of **AbstractErrorPolicyUser** to get the common implementation. As I will demonstrate later, it will often be useful to extend **ResultSetFunction** using an anonymous class. Anonymous classes cannot inherit from more than one class or interface, therefore I have written an `AbstractResultSetFunction` class that does nothing other than extend **AbstractErrorPolicyUser** and implement **ResultSetFunction**:

```
public abstract class AbstractResultSetFunction<T>
   extends AbstractErrorPolicyUser
   implements ResultSetFunction<T>
{}
```

Now all clients have to do is extend **AbstractResultSetFunction** and implement the read method.

The read method takes a **ResultSet** and returns a value. It is responsible for extracting the results from the result set and processing them into something that can be returned. As read does not have an exception specification it is also responsible for error handling. For example,

```
private static class Value<T>
   extends AbstractErrorPolicyUser
   implements ConnectionValue<T>
{
  private final String sql;
  private final ResultSetFunction<T> rsUser;
  public Value(String sql,
     ResultSetFunction<T> rsUser)
  {
    this.rsUser = rsUser;
    this.sql = sql;
  }
  @Override
  public T fetch(Connection con)
  {
    return new StatementProvider(
    con,getErrorPolicy()).provideTo(
    new ExecuteQuery<T>(sql,rsUser) );
  }
}
```

Listing 16

```
new AbstractResultSetFunction<String>()
{
  @Override
  public String read(ResultSet rs)
  {
    String result = null;
    try
    {
      if (rs.next())
      {
        result = rs.getString("url");
      }
    }
    catch(SQLException ex)
    {
      getErrorPolicy().handleError(ex);
    }
    return result;
  }
}
```

Listing 17

retrieving a single string from a database table (Listing 17) or retrieving multiple strings from a database table (Listing 18).

Putting it all together

To use the boilerplate to query a database, a client must first create a connection factory:

```
final ConnectionFactory conFactory =
  new StringConnection(DRIVER, CONNECTION_STRING)
  .setUser(USERNAME, PASSWORD)
  .setDatabase(DATABASE);
```

Then create a connection provider and pass it the connection factory and, optionally, a custom error policy:

```
final ConnectionProvider cp =
   new ConnectionProvider(conFactory);
```

Once created the connection factory and connection provider can be used for any number of queries and therefore only need to be created once.

```
new AbstractResultSetFunction<List<String>>()
{
  @Override
  public List<String> read(ResultSet rs)
  {
    List<String> result =
       new ArrayList<String>();
    try
    {
      while (rs.next())
      {
        result.add(rs.getString("url"));
      }
    }
    catch(SQLException ex)
    {
      getErrorPolicy().handleError(ex);
    }
    return result;
  }
}
```

Listing 18

```
final String s =
  Query.execute( cp,
     "select url from services where name =
     'Instruments'",
     new AbstractResultSetFunction<String>()
     {
    @Override
    public String read(ResultSet rs)
    {
      String result = null;
      try
        {
          if (rs.next())
          {
             result = rs.getString("url");
          }
        }
        catch(SQLException ex)
        {
          getErrorPolicy().handleError(ex);
        }
        return result;
      }
    });
```

Listing 19

Executing a query that does not return any results can then be done with a single statement:

```
Query.execute( cp,
  "insert into services ([name],[url])" +
  "VALUES(
  'Log','http://prodserv01/axis/services/Log')");
```

Queries that return results only take a little more, almost all of which is the anonymous class that processes the results from the result set (Listing 19).

Conclusion

Boiler plating database resource cleanup with Execute Around Method offers a high level of safety and encapsulation while not compromising on control of the database resources in client code. Reducing the amount of code that has to be written each time also reduces the possibility of mistakes and resource leaks. ■

Acknowledgements

Thank you to Adrian Fagg for guidance into Execute Around Method, not to mention quite a bit of help with the design.

References

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

[CheckedExceptions] 'Checked Exceptions' http://en.wikipedia.org/wiki/Exception_handling#Checked_exceptions

[CVu] *CVu*: http://accu.org/index.php/journals/c77/

[GoF] Gang of Four: *Design Patterns : Elements of reusable object-oriented software* by Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Addison Wesley ISBN-13: 978-0201633610

[Part I] 'Boiler Plating Database Resource Cleanup' – Part I: http://www.marauder-consulting.co.uk/Boiler_Plating_Database_Resource_Cleanup_-_Part_I.pdf

ACCU 2009

The 2009 ACCU Conference took place in March. The confernece chair, Giovanni Asproni, provides a report.

The conference is over, and I've fully recovered – being the conference chair is always very rewarding but also quite exhausting, even when the conference goes smoothly (and, perhaps, staying up until late drinking beers and chatting with the various delegates and speakers had an impact on that as well).

Despite the current economic climate, the attendance was excellent – there were about as many people as in 2008, and this at a time when several conferences lost a big chunk of their size and others, like the Software Development conferences series in the US, have been discontinued. In hindsight, this is not so surprising: many speakers and delegates told me they thought the programme was excellent. In a sense it was easy to achieve that, since we received a great number of high quality proposals, but it was also very difficult, because we could accept less than 50% of the proposals received due to the limited number of slots available, leaving out some very good ones.

The programme was also more intense than in previous years, with two keynotes on the Wednesday, and two lightning talks sessions, one on the Thursday and the other on the Friday.

The two Wednesday keynotes caused a bit of controversy – the first one at the start of the day, from Robert Martin, 'The Birth Of Software Craftsmanship', was followed at the end of the day by Nico Josuttis' 'Welcome Crappy Code: The Death of Code Quality'. Fortunately, the debate afterwards was very civilized, even if a bit heated – Bob Martin wrote a blog entry about that [Martin] and Nico wrote his own thoughts [Josuttis].

The lightning talks proved to be funny, informative, and were very well received – and Kevlin surprised many of us by proving that he could actually express some interesting ideas in five minutes only, without running over time.

Some other highlights include the engaging keynote from Baroness Susan Greenfield, 'Geeks, Nerds and Other Prejudices' – in which she debunked some myths on gender differences – which was received with an huge ovation (we had to ask people to stop applauding because we were running late!) – and the money collection for Bletchley Park through a variety of means: a raffle, voluntary donations (Astrid even convinced some other guests of the conference hotel to donate money for the cause), and the Enigma competition. Thanks to the generosity of our donors we managed to collect a total of £1400 which has already been given to the museum.

The success of the conference was due to many people, and I want to thank them all: the conference committee members; the people from Archer Yates, Julie, Marsha, and Charlotte, who, as usual, did an outstanding job; the sponsors, who allow us to keep the fees affordable; and all the speakers and attendees who always make this conference unique. I also want to give some very special thanks to a couple of friends, Allan Kelly and Kevlin Henney, who were always available to share ideas and opinions with me during the organization of the conference, and to Linda Rising, who kindly accepted to deliver a keynote in place of Frank Buschmann, who, unfortunately, had to pull out at the very last minute.

We are now working to the next edition with the aim of making the conference even better. If you have any suggestions, or are thinking of sending a proposal (and, perhaps, you want to check if your idea is potentially interesting or not), feel free to send me an email at conference@accu.org – the Call for Proposals is not out yet, but early submissions and ideas are always welcome.

References

[Josuttis] http://www.josuttis.de

[Martin] http://blog.objectmentor.com/articles/2009/04/23/crap-code-inevitable-rumblings-from-accu

Giovanni Asproni is the ACCU Conference Chair. If you have an idea for next year's conference, or just want to give some feedback for this year, he can be reached at conference@accu.org